SFA Modernization Partner

United States Department of Education
Student Financial Assistance

ITA Reusable Common Services

Build & Test Report

Task Order #46

Deliverable # 46.1.4

Version 1.0

September 27, 2001

Table of Contents

1 Introduction

1.1 Purpose

This Build and Test Report documents the build procedures, test conditions, and results of the ITA Release 2.0 Reusable Common Services (RCS). Specifically, this report provides:

- RCS test conditions and results
- RCS performance analysis
- RCS usage scenarios
- RCS repository and build approach

ITA Release 2.0 RCS includes:

- Component Factory Framework
- Email Framework
- Exception Handling Framework
- Logging Framework
- Persistence Framework
- Search Framework.

1.2 Approach

1.2.1 Testing

To ensure the quality of RCS, each service went through extensive unit testing. ITA relied on two methods in conducting unit tests: automated and manual unit test. Wherever applicable, ITA utilized JUnit for automated testing. JUnit is a set of Java packages that allows developers to readily create Java test cases for Java classes, and to then run these unit tests interactively or in batch mode. The intended result is higher quality code, as well as avoidance of the cumbersome and repetitive task of going back and reconstructing unit tests after all code has been written.

For special cases where automated unit tests could not be performed by JUnit, ITA developers conducted manual tests. Test conditions for automated and manual testing are provided in this document and expected results are also provided for manual testing.

Benefits to the unit test approach are:

- Standardized test conditions and cycles
- Increased code quality
- Increased consistency in the approach to testing
- Increased productivity
- Reduced time for regression testing
- More time available to spend on enhancements as less time is required for fixes

Test conditions are documented in a tabular format with the following column headings:

Column	Description
Condition Number	Test condition number
Detailed Condition	Detailed description of the test condition / case
Test Class Name	Test class name in JUnit Java code
Test Class Method	Test method (test case) in the test class
Class Name	Specific Java source file of the RCS code to be tested by the JUnit test code
Method Name	Specific method in the Java source file of the RCS code to be tested by the JUnit test code
Results	Expected results
Data File Name	Any configuration and / or data files needed to execute this test case

1.2.2 Performance Analysis

To ensure program efficiency and to detect possible bottlenecks, ITA used JProbe to analyze each RCS component to identify common performance issues such as thread integrity and memory management.

Common thread integrity issues include:

- Data Race Conditions – Occurs when concurrent threads attempt to access a shared resource at the same time. One thread can be writing to a shared resource at the same time another thread attempts to readfrom or write to the same shared resource. This will result in unreliable data.
- Deadlocks – Occurs when one thread is holding a lock while attempting to acquire a lock held by another thread, while at the same time, the second thread needs the lock held by the first. Incorrect programming logic will cause the threads to never move forward and cause the program to terminate.

Common memory management issues include:

- Loitering objects – Occurs when the application will not use the objects again, but the developers fail to remove the reference to the objects, the objects will remain, or loiter, in memory indefinitely. This condition can consume a significant amount of memory and degrade the Java Virtual Machine (JVM) performance.
- Excessive object allocation – If the application creates an excessive number of objects, the Java heap (a type of memory) will grow larger and garbage collection activities will take longer because there are more objects to evaluate. This will also degrade the JVM performance.

The performance analysis for each service is documented in this report. The topics included in the performance analysis are:

- Background information
- Test harness design
    - Test environment
    - Test configuration
    - Test scenario
- Memory (Heap) analysis
- Performance analysis
- General Performance Metrics

1.2.3 Configuration Management

ITA uses Rational ClearCase for its configuration management system. ClearCase manages multiple variants of evolving software systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies.

These capabilities enable ClearCase to address the critical requirements of organizations that produce and release software:

- **Effective Development:** ClearCase enables developers to work efficiently, allowing them to fine-tune the balance between sharing each other's work and isolating themselves from destabilizing changes. ClearCase automatically manages the sharing of both source files and the files produced by a software build.

- **Effective Management:** ClearCase tracks the software build process, so that developers can determine what was built, and how it was built. Further, ClearCase can instantly recreate the source base from which a software system was built, allowing it to be rebuilt, debugged, and updated all without interfering with other programming work.

- **Enforcement Of Development Policies:** ClearCase enables project administrators to define development policies and procedures, and to automate their enforcement.

1.2.4 ClearCase and Reusable Common Services

ClearCase is a robust version control system that can manage large projects with highly interdependent code. There are two main capabilities that RCS will utilize within ClearCase:

- **Version Control:** Developers use ClearCase on a daily basis to maintain a complete history of their project files. This will aid developers, project managers, and build managers maintain a complete picture of the progression of a resource.

- **Common Directory Structure:** A common directory structure will be implemented and utilized for each common service. This standard will simplify the process of releasing major and minor versions.

The RCS directory structure in ClearCase is as follows:

Directory	Function	Use
/resources	To maintain any configuration files that are utilized by the common services.	Developers should use this folder for any files that contain configuration information that is necessary for a common service to function at runtime.
/classes	To hold the compiled class files for the contents of the /src directory.	When ANT completes a build, this directory will be populated with the compiled source code. This directory should not be archived and versioned within ClearCase, for two reasons: 1) This will dramatically increase the size of the repository; 2) The class files are always available for use by compiling them from the source files.

Directory	Function	Use
/dist	To maintain a record of previous distribution .exe files.	When a major or minor release is completed, the corresponding distribution packages will be copied into this directory and added to version control. The same label that is applied to all files that make up a build will also be added to the .exe files.
/doc /internal/ /external/	To maintain any (non-javadoc) documentation both internal and external.	Documentation will be created during the design, development, and testing phases of the project. Testing conditions and other internal documentation will be placed within the internal folder, while user guides and design documentation will reside in the external folder.
/examples	Example programs that show how to properly use the common services and give the end user a code reference for their use.	Developers and testers will use this directory to store and version different example programs and scripts. There will be a directory for each common service in /examples and it is within these folders that the code will reside.
/javadoc	To provide an area for compiled javadocs to reside within the view and the distribution.	When developers and build managers make the distributions for RCS, this directory will be the output destination for the javadoc. There will be a directory for each common service within /javadoc. There will be no version control applied to these files, as they are derived from comments within the source code. This source code is already versioned within ClearCase, and there is no use in duplicating these files.
/lib	To place all third-party libraries that are necessary for a common service.	If developers need to use libraries that are not a part of the java standard library, they should place them in the /lib folder for their use. As there will be no change to these libraries, there is no need to include them in the ClearCase repository.
/src /gov/ed/sfa/ita/	To place source code and maintain the package structure for common services.	This is the main development area and it holds the bulk of the source code necessary for the common services. These folders adhere to the RCS package structure and are included when builds and distributions are made.

Directory	Function	Use
/test	To store automated unit test code for development and performance testing.	Within a common service folder is the versioned source code for testing each RCS component. These programs will be provided with the source code and full distribution for testing.
/build	To maintain and version the build files, batch files, and scripts that are necessary to build individual RCS components as well as the full complement.	This folder will contain the versioned files necessary to build each component and will be used almost solely by the build manager.

1.2.5 Build Management

ANT is a Java-based build tool used for RCS Build Management. ANT is an open-source project from the Jakarta Project. It is a powerful scripting tool that lets developers execute the build processes around the code requirements using predefined tasks. A defined build process ensures that the software in the development project is built in the exact same manner each time a build is executed. As the build process becomes more complex it becomes increasingly necessary to achieve such consistency. ANT is a platform-independent scripting tool that lets developers construct build scripts using a large number of built-in tasks with minimal customization.

The table below lists some of the major tasks that are built into the Ant distribution.

Command	Description
Ant	Used to execute another ant process from within the current one.
Copy	Used to copy directories and files
Copyfile	Used to copy a single file.
Cvs	Handles packages / modules retrieved from a CVS repository.
Delete	Deletes either a single file or all files in a specified directory and its sub-directories.
Deltree	Deletes a directory with all its files and subdirectories.
Exec	Executes a system command. When the os attribute is specified, then the command is only executed when Ant is run on one of the specified operating systems.
Get	Gets a file from an URL.
Jar	Jars a set of files.
Java	Executes a Java class within the running (Ant) VM or forks another VM if specified.
Javac	Compiles a source tree within the running (Ant) VM.
Javadoc	Generates code documentation using the javadoc tool.
Junit	Part of the Ant optional Tasks. Runs JUnit tests.

Command	Description
Mkdir	Makes a directory.
Property	Sets a property (by name and value), or set of properties (from file or resource) in the project.
Rmic	Runs the rmic compiler for a certain class.
Tstamp	Sets the DSTAMP, TSTAMP, and TODAY properties in the current project.
Style	Processes a set of documents via XSLT.
Zip	Zips a set of files

The ANT tool builds the following distribution packages for each RCS framework:

Package	Description
Distribution Package	- Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.package.exe - Contents: Executables, Source, Documentation, Release Notes
Executable	- Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.exe The Executable package will be the distribution for any project that would just like the deliverable components, with the necessary libraries and class files present. The self-extracting WinZip file will only need to be extracted into the class path of the application for the RCS components to be utilized.
Source	- Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.src.exe This is the package of all of the source files.
Documentation	- Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.doc.exe The documentation package will contain the design documents, user guides, examples, configuration files, test documentation and the JavaDoc associated with a particular common service.
Release Notes	- Package naming format: EX. readme.<Service>.<Major #>.<Minor #>.txt The release notes will be used to provide the end user of RCS components with an overview of the capabilities and any installation and build instructions in order to utilize the components.

1.2.6 RCS Source Code File Listing

The table below shows the list of Java source files for each of the RCS frameworks.

RCS	Source File
Component Factory	SFAFactory.java SFALocalProducer.java SFAObject.java SFAProducer.java SFASerializableObject.java
Email	SFASmtpClient.java
Exception	SFAException.java SFAExceptionFactory.java
Logging	Syslog.java

RCS	Source File
Persistence	ISFAPersistableMapper.java SFADomain.java SFAOracleParser.java SFAParameter.java SFAParser.java SFAPersistableObjectmanager.java SFAPersisConstants.java SFAQueryParser.java SFAResultSet.java SFAUnitofWork.java
Search	AutonomyConnection.java AutonomyResultSet.java AutonomyStatement.java

2 RCS – Component Factory Framework

The Component Factory, also known as the Object Factory, is a producer of objects. The ITA custom Component Factory incorporates the following:

- Support of applications running on the IBM WebSphere Application Server (WAS)
- Limit of the framework only to local classes since SFA applications are not implementing EJBs in their applications
- Integration with the ITA exception handling framework
- Integration with the ITA logging framework

The component factory encapsulates object creation logic by providing an instance of an object and not revealing its implementation. Implementing the component factory will provide application teams with the following benefits:

Rapid development and code reuse

If an environment needs to be configured before creating an instance of an object, then using the component factory will ease the object creation by allowing a developer to call the produce method of the factory and not to worry about setting up an environment. A properties file that is used to create the object will configure the environment. This will help promote rapid software development and code reuse.

Complex object creation

If object creation is complex (i.e., a class uses its subclasses to specify which objects it creates) then using the component factory can ease the development effort.

Migrating to different environments

If the production and development environments are not a mirror image of each other, then the developer can encapsulate all the configuration information in the SFAFactory class and just create an instance of an object. As a result, the component factory enables the definition of clear migration strategies from one architectural approach to another.

Technology change

The technology underlying the creation of an object can change. Future releases of RCS will extend the component factory to support EJBs and JDBC 2.0 DataSources. After the component factory is extended it can be used to create lite-EJBs that are collocated with servlets and JSPs. Later these lite-EJBs can be converted into complete EJBs and deployed onto an application server without changing any of the previously defined servlets and JSPs.

Arbitrary data types
The component factory can be used with arbitrary data types. The component factory is organized in such a way that it can instantiate other classes without being dependent on any of the classes it instantiates.

Adding new classes
The component factory can easily add new subclasses without impacting the existing classes. Thus, as the application gets more complex (e.g., the application uses EJBs) then the developer can easily extend the component factory by additional classes that create different types of objects.

2.1 Testing Conditions & Results

2.1.1 Automated Testing

Normal Conditions

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Load the ita properties file	TestComponentFactory	testloadProperties	SFAFactory	loadFile	Properties file loaded	rcs.xml, ita.properties, errormessage_US.properties
2	Set the environment	TestComponentFactory	testSetEnvironment	SFAFactory	setEnvironment	Environment is set	rcs.xml, ita.properties, errormessage_US.properties
3	Create an instance of the using default producer	TestComponentFactory	testProduce	SFAFactory	produce	A default object is created	rcs.xml, ita.properties, errormessage_US.properties
4	Create an instance of an object using local producer	TestComponentFactory	testProduce	SFAFactory	produce	Requested object is produced	rcs.xml, ita.properties, errormessage_US.properties

Exception Conditions

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Cannot load the ita properties file	TestComponentFactory	testLoadProperties	SFAFactory	loadFile	Exception is caught	rcs.xml, ita.properties, errormessage_US.properties
2	Cannot set the environment	TestComponentFactory	testSetEnvironment	SFAFactory	setEnvironment	Exception is caught	rcs.xml, ita.properties, errormessage_US.properties
3	The requested object is not produced	TestComponentFactory	testProduceDefault	SFAFactory	produce	Exception is caught	rcs.xml, ita.properties.fails, errormessage_US.properties

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
4	The requested object is not produced	TestComponentFactory	testProduceLocal	SFAFactory	produce	Exception is caught	rcs.xml, ita.properties.fails, errormessage_US.properties

2.2 Performance Analysis

2.2.1 Summary

This section reviews the performance testing of the ITA Component Factory framework. Its purpose is to provide an architectural review of the framework and identify potential performance issues and performance considerations that an application development group should be aware of when using the framework.

A thorough performance review of the ITA Component Factory framework revealed that this framework performs very well, using objects judiciously and using memory responsibly.

The test did not uncover deficiencies in the Component Factory framework itself. Instead, it showed that the framework is fast and lightweight, though framework configurations and improper use of the Logging framework might lead to slow performance. This behavior would be the result of formatting or logging messages in ways that are resource intensive or involve subsystem latencies. These concerns however are outside the scope of the Component Factory.

2.2.2 Test Environment

The testing harness was run on a standard SFA developer workstation. The hardware consisted of a Compaq Deskpro with a single 600 MHz Pentium III processor and 512 MB of RAM. The machine ran Windows NT 4.0 Service Pack 6. The Java environment was Sun's JDK 1.3. While tests were run, no other applications were loaded into memory, and the system was not interacted with. This was done in order to leave all resources available to the test harness, and eliminate the possibility of unexplained behavior in the tables of results.

2.2.3 Test Configuration

The ITA Component Factory framework was configured in a very standard manner, as it would be in actual usage. The configuration of the framework implements Logging and Exception frameworks. Therefore, any configurations these frameworks require were used.

2.2.4 Testing Scenarios

This Component Factory performance test focused on one usage scenario for its analysis: The creation of a Local Factory Producer defined by this property file:

```
ita.factory.producer.default=gov.ed.sfa.ita.componentfactory.SFALocalProducer
ita.factory.env.producer.default.classname=gov.ed.sfa.ita.componentfactory.RCS
ita.factory.producer.localProducer=gov.ed.sfa.ita.componentfactory.SFALocalPro
ducer
ita.factory.target.producer.alpha=localProducer
ita.factory.env.target.alpha.classname=gov.ed.sfa.ita.componentfactory.SFASeri
alizableObject
```

Here is the testing code:

```
package gov.ed.sfa.ita.componentfactory;

import java.io.*;
import gov.ed.sfa.ita.logging.Syslog;
import gov.ed.sfa.ita.exception.SFAException;
import gov.ed.sfa.ita.exception.SFAExceptionFactory;
import gov.ed.sfa.ita.exception.SFATrace;

public class TestComponentFactory {
      public TestComponentFactory() {
      }

      public static void main(String[] args) throws SFAException {

          System.out.println("Starting main...");
             Syslog.addLogging();
             String homedir = System.getProperty("user.home");

            java.util.Properties p = new java.util.Properties();
            File configfile = new File(homedir, "ita.properties");

            try {
                  p.load(new FileInputStream(configfile));
            } catch (IOException e) {
                  Syslog.log(
                        TestComponentFactory.class,
                        "RCS_CFactory",
                        null,
                        "Could not load the component factory properties
file\n",
                        Syslog.ERROR);
            }

            gov.ed.sfa.ita.componentfactory.SFAFactory.setEnvironment(p);

            gov.ed.sfa.ita.componentfactory.RCS defaultobj =
                  (gov.ed.sfa.ita.componentfactory.RCS)
SFAFactory.produce("default1");

            if (defaultobj != null) {
                  Syslog.log(
                        TestComponentFactory.class,
                        "RCS_CFactory",
                        "Requested object is produced.",
                        defaultobj,
                        Syslog.INFO);
            } else {
                  Syslog.log(
                        TestComponentFactory.class,
                        "RCS_CFactory",
                        null,
                        "Requested object is null - need more investigation.",
                        Syslog.ERROR);
            }

            gov.ed.sfa.ita.componentfactory.SFASerializableObject obj =
                  (gov.ed.sfa.ita.componentfactory.SFASerializableObject)
SFAFactory.produce(
                        "alpha");

            if (obj != null) {
                  Syslog.log(
                        TestComponentFactory.class,
                        "RCS_CFactory",
                        "Requested object is produced.",
                        obj,
                        Syslog.INFO);
            } else {
                  Syslog.log(
                        TestComponentFactory.class,
                        "RCS_CFactory",
                        null,
                        "Requested object is null - need more investigation.",
                        Syslog.ERROR);
            }

      }
}
```

2.2.5 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: Identifies any performance bottleneck due to inefficient code algorithms.

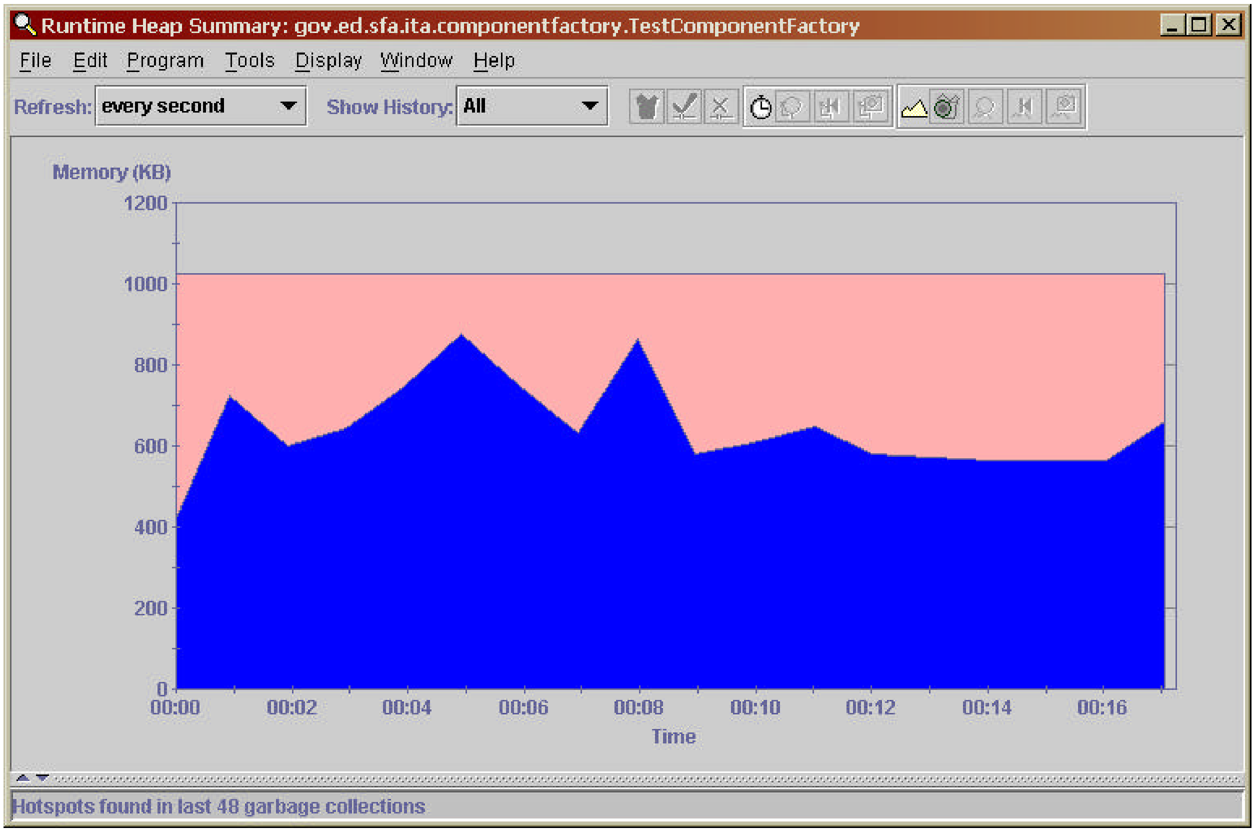
2.2.5.1 Memory (Heap) Usage

The performance test utilized JProbe Profiler’s Memory Debugger to identify the parts of the Component Factory framework that might be causing loitering objects. This was accomplished by analysis of the Java heap. The Runtime Heap Summary window can be used to view instance counts and information on allocating methods.

The Heap Usage Chart below plots the size of the Java heap with a time interval of 1 second. The chart visualizes memory use within the Java heap. It displays the available size of the Java heap (the light gray line above 2000 KB) and the used memory (the dark lines with peaks) over time.

The heap usage chart shows a series of spikes. Steep spikes in the Heap Usage Chart represent temporary objects being allocated and then garbage collected. If the levels of the troughs become higher over time, then not all the temporary objects are garbage collected. Even though multiple objects are being created and destroyed, the troughs remain steady over time. As a result, there do not appear to be any lingering objects.

Runtime Heap Usage


Runtime Heap Summary: gov.ed.sfa.ita.componentfactory.TestComponentFactory
File Edit Program Tools Display Window Help
Refresh: every second
Show History: All
Memory (KB)
1200
1000
800
600
400
200
0
00:00
00:02
00:04
00:06
00:08
00:10
00:12
00:14
00:16
Time
Hotspots found in last 48 garbage collections

2.2.5.2 Garbage collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage-collected objects accumulates. The list below is the final top three, displaying cumulative objects created at the end of program execution.

Each row identifies the class by package name and class name. The next columns state, in order, the number of garbage collected objects (GC'ed column) for the class, the number of instances remaining in the heap (Alive column), and the method that allocated the instances of the class (AllocatedAt column). The same class can appear more than once because more than one method allocated instances of the class.

The chart below shows expected activity and does not show anything that would indicate a performance problem. These numbers are in line with the framework requirements and expected behavior.

Garbage Collection Statistics

Package	Class	GC'ed	Alive	Allocated At
gov.ed.sfa.ita.exception	SFATrace	1,974	1	gov.ed.sfa.ita.componentfactory.SFALocalProducer.produce
gov.ed.sfa.ita.componentfactory	SFASerializableObject	986	1	java.lang.Class.newInstance0
gov.ed.sfa.ita.componentfactory	RCS	986	1	java.lang.Class.newInstance0

2.2.5.3 Code Efficiency

There are nine efficiency metrics that can be collected in JProbe — five basic metrics and four compound metrics. The basic metrics include Number of Calls, Method Time, Cumulative Time, Method Object Count, and Cumulative Object Count. The compound metrics are averages per number of calls, including Average Method Time, Average Cumulative Time, Average Method Object Count, and Average Cumulative Object Count. Time is measured as CPU time.

The following list defines the nine performance metrics:

- Number of Calls - The number of times the method was invoked.
- Method Time - The amount of time spent executing the method, excluding time spent in its descendants.
- Cumulative Time - The total amount of time spent executing the method, including time spent in its descendants but excluding time spent in recursive calls to descendants.
- Method Object Count - The number of objects created during the method’s execution, excluding those created by its descendants.
- Cumulative Object Count - The total number of objects created during the method’s execution, including those created by its descendants.
- Average Method Time - Method Time divided by Number of Calls.
- Average Cumulative Time - Cumulative Time divided by Number of Calls.
- Average Method Object - Count Method Object Count divided by Number of Calls.

The charts on the following pages serve to document the performance characteristics of the factory framework with lists based on the above metrics:

- Number of Calls
- Average Cumulative Time
- Average Method Time
- Average Cumulative Objects
- Average Method Objects

These measures are basic indicators of processing resource utilization. The lists can be reviewed for unexpected activity or optimization opportunities.

Methods with the most calls:

Number of Calls

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAFactory.produce(String)	2,000	3,846.2 (0.0%)	2.7 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
SFAFactory.produce(String, Object[])	2,000	3,843.5 (0.0%)	11.2 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
SFALocalProducer.produce(String, Object[], Properties)	2,000	2,795.8 (0.0%)	33.4 (0.0%)	82,639 (48.3%)	41 (0.0%)	1 (0.0%)
SFATrace.<init>(Object, String)	2,000	2,644.4 (0.0%)	26.7 (0.0%)	77,079 (45.1%)	38 (0.0%)	1 (0.0%)
SFATrace.getTrace()	2,000	2,394.7 (0.0%)	78.7 (0.0%)	61,007 (35.7%)	30 (0.0%)	4 (0.0%)
Throwable.printStackTrace(PrintWriter)	2,000	1,749.9 (0.0%)	1,748.7 (0.0%)	37,003 (21.6%)	18 (0.0%)	18 (0.0%)
Syslog.debug(Object, Object)	2,000	120.2 (0.0%)	120.2 (0.0%)	6,070 (3.6%)	3 (0.0%)	3 (0.0%)
PrintWriter.<init>(Writer)	2,000	62.0 (0.0%)	62.0 (0.0%)	2,000 (1.2%)	1 (0.0%)	1 (0.0%)
SFAException.<init>()	2,000	40.3 (0.0%)	3.2 (0.0%)	4,000 (2.3%)	2 (0.0%)	0 (0.0%)
Exception.<init>()	2,000	37.1 (0.0%)	37.1 (0.0%)	4,000 (2.3%)	2 (0.0%)	2 (0.0%)
StringWriter.<init>()	2,000	34.2 (0.0%)	34.2 (0.0%)	4,000 (2.3%)	2 (0.0%)	2 (0.0%)
StringWriter.toString()	2,000	33.8 (0.0%)	33.8 (0.0%)	2,000 (1.2%)	1 (0.0%)	1 (0.0%)
SFAExceptionFactory.getInstance()	2,000	26.0 (0.0%)	1.4 (0.0%)	1,202 (0.7%)	0 (0.0%)	0 (0.0%)
SFAFactory.init()	2,000	24.0 (0.0%)	1.8 (0.0%)	627 (0.4%)	0 (0.0%)	0 (0.0%)

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAProducer.getProperty(String, Properties)	2,000	20.9 (0.0%)	3.6 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SFAFactory.getProducerContainer(String)	2,000	12.8 (0.0%)	3.9 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
StringTokenizer.hasMoreTokens()	2,000	11.7 (0.0%)	11.7 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
String.valueOf(Object)	2,000	3.3 (0.0%)	3.3 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SFAException.getMessage()	2,000	1.2 (0.0%)	1.2 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SFAFactory$ProducerContainer.produce(String, Object[])	1,000	2,821.3 (0.0%)	3.1 (0.0%)	42,097 (24.6%)	42 (0.0%)	0 (0.0%)
RCS.<init>()	1,000	3.8 (0.0%)	2.0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SFASerializableObject.<init>()	1,000	2.9 (0.0%)	2.0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SFAFactory.setEnvironment(Properties)	1,000	1.6 (0.0%)	1.6 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

Average total time (includes time spent in sub-methods) spent in each method:

Average Cumulative Time

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAExceptionFactory.<init>()	1	49,028.2 (0.1%)	121.9 (0.0%)	1,201 (0.7%)	1,201 (0.7%)	4 (0.0%)
SFAFactory.initProducers()	1	29,856.8 (0.1%)	264.3 (0.0%)	413 (0.2%)	413 (0.2%)	4 (0.0%)
SFAFactory.doRegisterProducer(String, String)	2	11,966.4 (0.0%)	83.7 (0.0%)	354 (0.2%)	177 (0.1%)	2 (0.0%)
SFAFactory.initTargetProducers()	1	11,512.0 (0.0%)	120.9 (0.0%)	178 (0.1%)	178 (0.1%)	2 (0.0%)
SFAFactory.doRegisterTarget(String, String)	1	8,160.1 (0.0%)	125.7 (0.0%)	127 (0.1%)	127 (0.1%)	4 (0.0%)
SFAFactory.addTargetProducer(String, String, Properties)	1	5,908.8 (0.0%)	864.1 (0.0%)	99 (0.1%)	99 (0.1%)	3 (0.0%)
ClassLoader.loadClassInternal(String)	32	5,397.8 (0.0%)	5,397.8 (0.0%)	1,216 (0.7%)	38 (0.0%)	38 (0.0%)
SFAFactory.produce(String)	2,000	3,846.2 (0.0%)	2.7 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
SFAFactory.produce(String, Object[])	2,000	3,843.5 (0.0%)	11.2 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
Properties.load(InputStream)	1	3,813.8 (0.0%)	3,813.8 (0.0%)	69 (0.0%)	69 (0.0%)	69 (0.0%)
SFAFactory.getDefaultEnvironment()	1	3,157.4 (0.0%)	173.1 (0.0%)	34 (0.0%)	34 (0.0%)	5 (0.0%)
SFAFactory.doAddTargetProperty(String, Hashtable)	1	2,954.1 (0.0%)	138.0 (0.0%)	44 (0.0%)	44 (0.0%)	3 (0.0%)
SFAFactory.doAddProducerProperty(String, Hashtable)	1	2,833.9 (0.0%)	209.2 (0.0%)	48 (0.0%)	48 (0.0%)	7 (0.0%)
SFAFactory$ProducerContainer.produce(String, Object[])	1,000	2,821.3 (0.0%)	3.1 (0.0%)	42,097 (24.6%)	42 (0.0%)	0 (0.0%)

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFALocalProducer.produce(String, Object[], Properties)	2,000	2,795.8 (0.0%)	33.4 (0.0%)	82,639 (48.3%)	41 (0.0%)	1 (0.0%)
SFATrace.<init>(Object, String)	2,000	2,644.4 (0.0%)	26.7 (0.0%)	77,079 (45.1%)	38 (0.0%)	1 (0.0%)
SFATrace.getTrace()	2,000	2,394.7 (0.0%)	78.7 (0.0%)	61,007 (35.7%)	30 (0.0%)	4 (0.0%)

Average time spent within a method (not including sub-methods):

Average Method Time

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAFactory.addTargetProducer(String, String, Properties)	1	5,908.8 (0.0%)	864.1 (0.0%)	99 (0.1%)	99 (0.1%)	3 (0.0%)
PrintStream.println(String)	1	644.9 (0.0%)	644.9 (0.0%)	2 (0.0%)	2 (0.0%)	2 (0.0%)
Hashtable.keys()	5	497.8 (0.0%)	497.8 (0.0%)	6 (0.0%)	1 (0.0%)	1 (0.0%)
Syslog.addLogging()	1	1,270,697.9 (3.3%)	432.0 (0.0%)	13,240 (7.7%)	13,240 (7.7%)	16 (0.0%)
File.<init>(String, String)	2	292.7 (0.0%)	292.7 (0.0%)	16 (0.0%)	8 (0.0%)	8 (0.0%)
SFAFactory.initProducers()	1	29,856.8 (0.1%)	264.3 (0.0%)	413 (0.2%)	413 (0.2%)	4 (0.0%)

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
FileInputStream.<init>(File)	1	212.8 (0.0%)	212.8 (0.0%)	1 (0.0%)	1 (0.0%)	1 (0.0%)
SFAFactory.doAddProducerProperty(String, Hashtable)	1	2,833.9 (0.0%)	209.2 (0.0%)	48 (0.0%)	48 (0.0%)	7 (0.0%)
SFAFactory.getDefaultEnvironment()	1	3,157.4 (0.0%)	173.1 (0.0%)	34 (0.0%)	34 (0.0%)	5 (0.0%)
SFAFactory.doAddTargetProperty(String, Hashtable)	1	2,954.1 (0.0%)	138.0 (0.0%)	44 (0.0%)	44 (0.0%)	3 (0.0%)
SFAFactory.doRegisterTarget(String, String)	1	8,160.1 (0.0%)	125.7 (0.0%)	127 (0.1%)	127 (0.1%)	4 (0.0%)
StringTokenizer.nextToken()	6,000	125.6 (0.0%)	125.6 (0.0%)	6,000 (3.5%)	1 (0.0%)	1 (0.0%)
SFAExceptionFactory.<init>()	1	49,028.2 (0.1%)	121.9 (0.0%)	1,201 (0.7%)	1,201 (0.7%)	4 (0.0%)
SFAFactory.initTargetProducers()	1	11,512.0 (0.0%)	120.9 (0.0%)	178 (0.1%)	178 (0.1%)	2 (0.0%)
Syslog.debug(Object, Object)	2,000	120.2 (0.0%)	120.2 (0.0%)	6,070 (3.6%)	3 (0.0%)	3 (0.0%)

Average Cumulative Objects (includes objects created in sub-methods):

Average Cumulative Objects

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAExceptionFactory.<init>()	1	49,028.2 (0.1%)	121.9 (0.0%)	1,201 (0.7%)	1,201 (0.7%)	4 (0.0%)
ResourceBundle.getBundle(String)	1	48,834.4 (0.1%)	48,834.4 (0.1%)	1,197 (0.7%)	1,197 (0.7%)	1,197 (0.7%)
SFAFactory.initProducers()	1	29,856.8 (0.1%)	264.3 (0.0%)	413 (0.2%)	413 (0.2%)	4 (0.0%)
SFAFactory.initTargetProducers()	1	11,512.0 (0.0%)	120.9 (0.0%)	178 (0.1%)	178 (0.1%)	2 (0.0%)
SFAFactory.doRegisterProducer(String, String)	2	11,966.4 (0.0%)	83.7 (0.0%)	354 (0.2%)	177 (0.1%)	2 (0.0%)
SFAFactory.doRegisterTarget(String, String)	1	8,160.1 (0.0%)	125.7 (0.0%)	127 (0.1%)	127 (0.1%)	4 (0.0%)
SFAFactory.addTargetProducer(String, String, Properties)	1	5,908.8 (0.0%)	864.1 (0.0%)	99 (0.1%)	99 (0.1%)	3 (0.0%)
Properties.load(InputStream)	1	3,813.8 (0.0%)	3,813.8 (0.0%)	69 (0.0%)	69 (0.0%)	69 (0.0%)
SFAFactory.produce(String, Object[])	2,000	3,843.5 (0.0%)	11.2 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
SFAFactory.produce(String)	2,000	3,846.2 (0.0%)	2.7 (0.0%)	107,378 (62.8%)	53 (0.0%)	0 (0.0%)
SFAFactory.doAddProducerProperty(String, Hashtable)	1	2,833.9 (0.0%)	209.2 (0.0%)	48 (0.0%)	48 (0.0%)	7 (0.0%)
SFAFactory.doAddTargetProperty(String, Hashtable)	1	2,954.1 (0.0%)	138.0 (0.0%)	44 (0.0%)	44 (0.0%)	3 (0.0%)
SFAFactory$ProducerContainer.produce(String, Object[])	1,000	2,821.3 (0.0%)	3.1 (0.0%)	42,097 (24.6%)	42 (0.0%)	0 (0.0%)
SFALocalProducer.produce(String, Object[], Properties)	2,000	2,795.8 (0.0%)	33.4 (0.0%)	82,639 (48.3%)	41 (0.0%)	1 (0.0%)

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFATrace.<init>(Object, String)	2,000	2,644.4 (0.0%)	26.7 (0.0%)	77,079 (45.1%)	38 (0.0%)	1 (0.0%)
ClassLoader.loadClassInternal(String)	32	5,397.8 (0.0%)	5,397.8 (0.0%)	1,216 (0.7%)	38 (0.0%)	38 (0.0%)
SFAFactory.getDefaultEnvironment()	1	3,157.4 (0.0%)	173.1 (0.0%)	34 (0.0%)	34 (0.0%)	5 (0.0%)
SFATrace.getTrace()	2,000	2,394.7 (0.0%)	78.7 (0.0%)	61,007 (35.7%)	30 (0.0%)	4 (0.0%)

Average number of Objects within a method (not including submethods):

Average Method Objects

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFAFactory.doAddProducerProperty(String, Hashtable)	1	2,833.9 (0.0%)	209.2 (0.0%)	48 (0.0%)	48 (0.0%)	7 (0.0%)
SFAFactory.getDefaultEnvironment()	1	3,157.4 (0.0%)	173.1 (0.0%)	34 (0.0%)	34 (0.0%)	5 (0.0%)
SFAExceptionFactory.<init>()	1	49,028.2 (0.1%)	121.9 (0.0%)	1,201 (0.7%)	1,201 (0.7%)	4 (0.0%)
SFAFactory.initProducers()	1	29,856.8 (0.1%)	264.3 (0.0%)	413 (0.2%)	413 (0.2%)	4 (0.0%)
SFAFactory.doRegisterTarget(String, String)	1	8,160.1 (0.0%)	125.7 (0.0%)	127 (0.1%)	127 (0.1%)	4 (0.0%)

Name	Calls	Average Cumulative Time	Average Method Time	Cumulative Objects	Average Cumulative Objects	Average Method Objects
SFATrace.getTrace()	2,000	2,394.7 (0.0%)	78.7 (0.0%)	61,007 (35.7%)	30 (0.0%)	4 (0.0%)
.Thread-0.	1	1,405.8 (0.0%)	1,405.8 (0.0%)	4 (0.0%)	4 (0.0%)	4 (0.0%)
SFAFactory.addTargetProducer(String, String, Properties)	1	5,908.8 (0.0%)	864.1 (0.0%)	99 (0.1%)	99 (0.1%)	3 (0.0%)
SFAFactory.doAddTargetProperty(String, Hashtable)	1	2,954.1 (0.0%)	138.0 (0.0%)	44 (0.0%)	44 (0.0%)	3 (0.0%)
Syslog.debug(Object, Object)	2,000	120.2 (0.0%)	120.2 (0.0%)	6,070 (3.6%)	3 (0.0%)	3 (0.0%)
SFAFactory.initTargetProducers()	1	11,512.0 (0.0%)	120.9 (0.0%)	178 (0.1%)	178 (0.1%)	2 (0.0%)
SFAFactory.doRegisterProducer(String, String)	2	11,966.4 (0.0%)	83.7 (0.0%)	354 (0.2%)	177 (0.1%)	2 (0.0%)

2.3 User Guide

2.3.1 Introduction

2.3.1.1 Purpose

This section provides a high-level summary and usage of the Integrated Technical Architecture (ITA) standard Java component factory Framework. The component factory framework is part of a suite of frameworks provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects.

2.3.1.2 Intended audience

This section is intended for ITA and SFA application programmers who need to understand the ITA component factory framework in order to use this framework in their applications.

2.3.2 Background

The Component Factory, also known as the Object Factory, is a producer of objects. The ITA custom Component Factory incorporates the following:

- Support of applications running on the IBM WebSphere Application Server (WAS)
- Limit of the framework only to local classes since SFA applications are not implementing EJBs in their applications
- Integration with the ITA exception handling framework
- Integration with the ITA logging framework

2.3.2.1 Scope

This section covers installation, configuration, and features of the ITA component factory framework. This section contains information on SFALocalProducer. While the component factory uses the logging framework and the exception handling framework, these frameworks are not covered in this section. Consult the logging and exception handling user guide for more information on the logging and exception handling frameworks.

2.3.2.2 Assumptions

It is assumed that the component factory framework will function in a J2EE application server environment. As the current production server for SFA is IBM’s WebSphere 3.5, the framework will be compiled using its required JDK version 1.2.2. It should also work with the current JavaServer Pages (1.1), Java Servlet (2.2), Java Messaging Service (1.0.1), and Java Database Connectivity (2.0) specifications for this server.

2.3.3 Description

2.3.3.1 Overview

The component factory implements the factory method pattern. There is no single class that makes the decision as to which subclass to instantiate. Instead, the superclass (SFAProducer) defers the decision to each subclass. A program written using this pattern defines an abstract class that creates objects, but lets each subclass decide which object to create.

The component factory framework defines a general and extensible factory mechanism. It allows developers to completely decouple how objects and components are instantiated from their use. The ITA component factory framework offers a standardized approach to retrieving system components through a predefined lookup mechanism.

2.3.3.2 Main Concepts

The component factory framework is a producer of objects. It accepts some information about how to create an object, such as a reference, and then returns an instance of that object. Components must be registered with the component factory. In order to use the component factory the user needs to understand producers and targets.

Producers: Producers define the strategy to produce an object reference for a given key. Producers are named entities that produce objects based on configuration properties in the properties file.

Default Producer: A default producer may be defined by setting the property, sfa.factory.producer.default, to the desired producer class. If no default is set, then a producer is not used for the target and a warning should be logged. The ITA 2.0 Logging framework should be used for all logging purposes. A default producer is set through the properties as shown below.
sfa.factory.producer.default = <classname>

Targets: Targets logically represent what clients want to create. A named producer creates targets. More specific properties may be applied to each target. Producers use these properties to produce target instances. Typically, target properties override more general producer level properties with the same name, but this is dependent on the producer implementation (via the SFAProducer.getProperty() method).

2.3.4 Installation

2.3.4.1 Software requirements

The component factory framework is J2EE compliant. The framework requires JDK 1.2 (recommended). It also works with JDK 1.3.

Application Server: The ITA team will test the component factory framework in the WebSphere Application Server 3.5. There is nothing in the logging package that will tie the framework to a particular application server.

ITA Component Factory Package: The ITA component factory needs the following jar files:

- rcs_componentFactory_ v1.1.jar
- ita.properties

The ita.properties file allows developers to define the object they want to create it. The rcs_componentFactory_v1.0.jar depends on the ITA Logging and ITA Exception Handling frameworks. Developers should refer to the ITA Logging and ITA Exception Handling user guide for more details. The ITA logging framework requires following jar files:

- protomatter-1_1_5.jar
- jakarta-oro-2.0.1.jar
- jdom-B6.jar
- xerces.jar
- utility.jar
- xml.jar
- rcs_logging_v1.5.jar
- rcs.xml
- StartupRcs.jar

The ITA Exception Handling framework requires the following jar files:

- rcs_exceptionHandling_v1.5.jar
- errorMessages_en_US.properties

2.3.4.2 Installation procedures

1. Copy all of the above jar files into a directory (e.g. /www/dev/rcs/jars).

2. Copy rcs.xml, errorMessages_en_US.properties into a directory (eg. /www/dev/ita/properties).

3. Copy ita.properties into the user’s home directory (eg. root directory in the Solaris machine).

4. Add the Jar files on the classpath.

2.3.5 Configuration

The component factory package needs to be added in WebSphere’s classpath. The following steps show how to add the classpath on WebSphere.

a) Bring up the WebSphere admin console and select your application server on the console.

b) Stop your application server.

c) Click on the ‘General’ Tab and add the following line in the Command Line Arguments
- classpath /www/dev/rcs/jars/

d) Restart your application server.

Since the component factory uses the ITA logging and exception handling frameworks, the developer needs to refer to the logging and exception handling user guide to set up the framework correctly.

2.3.6 Usage Scenarios

The following two examples illustrate the use of the component factory.

2.3.6.1 Creating a default object

The following is an example of local component registration in the <application>.properties (e.g. ita.properties)file:

```
### Define a default Producer
ita.factory.producer.default=gov.ed.sfa.ita.componentfactory.SFAL
ocalProducer

###Define the object you want to create
ita.factory.env.producer.default.classname=gov.ed.sfa.ita.componentfactory.RCS
sfa.factory.producer.localProducer=gov.ed.sfa.ita.componentfactory.SFALocalProducer
```

Once the components are registered the program needs to lookup the component.

```
gov.ed.sfa.ita.TestComponentFactory obj =
(gov.ed.sfa.ita.TestComponentFactory)SFAFactory.produce( "default"
);

Syslog.log(TestComponentFactory.class, “RCS_Cfactory”, “Requested object is produced.” ,
obj, Syslog.INFO);
```

The sample program can be found at section 6. The output of the above fragment of code is as follows:

```
20:11:15 09/07 [INFO] TestComponentFactory Requested object is produced
gov.ed.sfa.ita.componentfactory.RCS
```

2.3.6.2 Creating a local object

The following example shows how to create an instance of a local object

```
### Define the local producer
ita.factory.producer.localProducer=gov.ed.sfa.ita.componentfactory.SFA
LocalProducer

### Alpha is a target-name and localProducer is producer name
ita.factory.target.producer.alpha=localProducer

###Define the object you want to create
ita.factory.env.target.alpha.classname=gov.ed.sfa.ita.componentfactory
.SFASerializableObject
```

Once the components are registered the program needs to lookup the component.

```
gov.ed.sfa.ita.TestComponentFactory obj2 =
(gov.ed.sfa.ita.TestComponentFactory)SFAFactory.produce( "alpha" );

Syslog.log(TestComponentFactory.class, “RCS_Cfactory”, “Requested
object is produced.” , obj2, Syslog.INFO);
```

The sample program can be found at section 6. The output of the above fragment of code is as follows:

```
20:11:15 09/07 [INFO] TestComponentFactory Requested object is produced
gov.ed.sfa.ita.componentfactory.SFASerializableObject
```

2.3.7 Sample Code

The following sample code shows different ways to create an object using the component factory.

```
package gov.ed.sfa.ita.componentfactory;


import java.io.*;
import gov.ed.sfa.ita.logging.Syslog;
import gov.ed.sfa.ita.exception.SFAException;
import gov.ed.sfa.ita.exception.SFAExceptionFactory;
import gov.ed.sfa.ita.exception.SFATrace;
/**
 * TestComponentFactory:  This class tests the component factory.  It
uses the ITA logging and exception handling framework
 * Creation date: (7/2/01 3:14:43 PM)
 * @author: Roshani Bhatt
 */
public class TestComponentFactory {
    public TestComponentFactory() {
    }
    /**
     * Creation date: (7/2/01 3:15:34 PM)
     * @param args java.lang.String[]
     */
    public static void main(String[] args) throws SFAException {
         //loads the logging configuration file
         Syslog.addLogging();
         String homedir = System.getProperty("user.home");

        java.util.Properties p = new java.util.Properties();
        File configfile = new File(homedir, "ita.properties");
        try {
            p.load(new FileInputStream(configfile));
        } catch (IOException e) {
            Syslog.log(
                TestComponentFactory.class,
                "RCS_CFactory",
                null,
                "Could not load the component factory properties
file\n",
                Syslog.ERROR);
        }
        gov.ed.sfa.ita.componentfactory.SFAFactory.setEnvironment(p);
        gov.ed.sfa.ita.componentfactory.RCS defaultobj =
            (gov.ed.sfa.ita.componentfactory.RCS)
SFAFactory.produce("default1");
        if (defaultobj != null) {

            Syslog.log(
                TestComponentFactory.class,
                "RCS_CFactory",
                "Requested object is produced.",
                defaultobj,
                Syslog.INFO);
        } else {

            Syslog.log(
                TestComponentFactory.class,
                "RCS_CFactory",
                null,
                "Requested object is null - need more investigation.",
                Syslog.ERROR);
        }
        gov.ed.sfa.ita.componentfactory.SFASerializableObject obj =
            (gov.ed.sfa.ita.componentfactory.SFASerializableObject)
SFAFactory.produce(
                "alpha");
        if (obj != null) {

            Syslog.log(
                TestComponentFactory.class,
                "RCS_CFactory",
                "Requested object is produced.",
                obj,
                Syslog.INFO);
        } else {

            Syslog.log(
                TestComponentFactory.class,
                "RCS_CFactory",
                null,
                "Requested object is null - need more investigation.",
                Syslog.ERROR);
        }
   } }
```

2.3.8 Resources

The following resources have more information about the component factory.

- RCS Component Factory Design Document

3 RCS – Email Framework

Previously, SFA applications used different methods to send e-mail messages. These methods included using a previous version of the JavaMail API and a proprietary design that retrieves messages out of an Oracle database, processes each message with UNIX shell scripts, and then sends the e-mail with the operating system utility sendmail.

The e-mail framework provides SFA with a common way to generate e-mail messages from applications. The e-mail framework uses Sun Microsystems' JavaMail API 1.2, which provides a standard interface for Java programs to send e-mails to a Simple Mail Transport Protocol (SMTP) Mail server. The intent is to supply an easy to use interface with JavaMail in SFA's applications.

3.1 Testing Conditions & Results

3.1.1 Automated Testing

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	The m_debug variable has been initilized	TestSFASmtpClient	testgetDebug	SFASmtpClient	getDebug	return variable	rcs.xml, errormessages. properties
2	The m_fileAttachments variable has been initilized	TestSFASmtpClient	testgetfileAttachments	SFASmtpClient	getfileAttachments	return variable	rcs.xml, errormessages. properties
3	the m_hostname has been initilized	TestSFASmtpClient	testgetHost	SFASmtpClient	getHost	return variable	rcs.xml, errormessages. properties
4	the m_SentDate has been initilized	TestSFASmtpClient	testgetSentDate	SFASmtpClient	getSentDate	return variable	rcs.xml, errormessages. properties
5	TextContent has been initilized	TestSFASmtpClient	testgetTextContent	SFASmtpClient	getTextContent	return variable	rcs.xml, errormessages. properties
6	The m_debug variable will be set to true and then checked	TestSFASmtpClient	testsetDebug	SFASmtpClient	setDebug	return variable	rcs.xml, errormessages. properties
7	The m_fileAttachments variable will be set and then checked	TestSFASmtpClient	testsetfileAttachments	SFASmtpClient	setfileAttachments	return variable	rcs.xml, errormessages. properties
8	the m_hostname will be set and then checked	TestSFASmtpClient	testsetHost	SFASmtpClient	setHost	return variable	rcs.xml, errormessages. properties
9	the m_SentDate will be set and then checked	TestSFASmtpClient	testsetSentDate	SFASmtpClient	setSentDate	return variable	rcs.xml, errormessages. properties
10	m_textContent will be set and then checked	TestSFASmtpClient	testsetTextContent	SFASmtpClient	setTextContent	return variable	rcs.xml, errormessages. properties

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
11	"From" string address=null	TestSFASmtpClient	testSFASmtpClientsendMessageStringfrom_is_null	SFASmtpClient	sendMessage(string)	The email is not sent and an SFAException is thrown	rcs.xml, errormessages.properties
12	"To" string address=null	TestSFASmtpClient	testSFASmtpClientsendMessageStringto_is_null	SFASmtpClient	sendMessage(string)	The email is not sent and an SFAException is thrown	rcs.xml, errormessages.properties
13	"Reply" string address=null	TestSFASmtpClient	testSFASmtpClientsendMessageStringreply_is_null	SFASmtpClient	sendMessage(string)	The email is not sent and an SFAException is thrown	rcs.xml, errormessages.properties

3.2 Performance Analysis

3.2.1 Summary

RCS Email framework is tested under a test harness using JProbe Profiler. Statistics collected is filtered to reflect the effects of SFASmtpClient class only. The test harness constructs and sends a simple email message using SFASmtpClient class. The heap analysis shows no loitering objects and the performance analysis suggests efficient coding. A general performance metrics is given as a comparison between email with and without attachment. The general performance metrics gives a rough idea how the framework performs.

3.2.2 Test Environment

The RCS Email framework performance test is conducted in JProbe. JProbe is a performance-profiling tool for Java based programs. Two key groups of statistics are given from the profiler: memory (heap) usage and time spent on each method within the program (performance). By using this performance-profiling tool, users can identify loitering objects and inefficiencies in code more easily. JProbe also has drill-down capability that allows gathering detailed information on individual methods and the interaction among them.

The performance test is done on Windows 2000 platform. The focus of the performance test is finding loitering objects and time spent on each method relative to each other. Even though the framework is envisioned to be used under a Unix environment, the test result gathered from Windows 2000 is transferable and valid because of the focus of the test. The hardware that is used to carry out the test is Toshiba Tecra 8100 with a PIII 800 MHz processor and 256 M RAM.

RCS Email framework is an API and needs to be tested under a harness. The statistics taken from the performance test excludes the harness by drilling down to the sendMessage() method.

Within the email framework, the SMTP server and the accompanying accessing information need to be supplied. The test is performed without this information since email severs used will be different for each application. As mentioned above, the focus of the test is to find loitering objects and locate inefficiencies within the code itself, the actual time spent on accessing the email server is out of the control of the framework and thus not tested.

3.2.3 Test Configuration

The Email framework uses the Logging framework as part of its reporting function. Messages are reported on the screen as well as logged in a designated file. Messages with severity level "Info" or above are logged. The logging configuration used for the performance test is as follows:

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <Syslog defaultMask="INFO" backgroundLogging="false">
- <Logger class="com.protomatter.syslog.FileLog" name="logging">
  <fileName>c:/gov/ed/sfa/ita/logging/test.log</fileName>
  <autoFlush>true</autoFlush>
  <stream>System.out</stream>
- <Policy class="com.protomatter.syslog.SimpleLogPolicy">
  <channels>ALL_CHANNEL</channels>
  <logMask>INHERIT_MASK</logMask>
  </Policy>
+ <Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
  </Logger>
- <Logger class="com.protomatter.syslog.PrintWriterLog"
name="PrintWriterLog.err">
  <autoFlush>true</autoFlush>
  <stream>System.out</stream>
- <Policy class="com.protomatter.syslog.SimpleLogPolicy">
  <channels>ALL_CHANNEL</channels>
  <logMask>INHERIT_MASK</logMask>
  </Policy>
- <Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
  <showChannel>false</showChannel>
  <showThreadName>false</showThreadName>
  <showHostName>false</showHostName>
  <dateFormat>HH:mm:ss MM/dd</dateFormat>
  <dateFormatCacheTime>1000</dateFormatCacheTime>
  <dateFormatTimeZone>America/New_York</dateFormatTimeZone>
  </Format>
  </Logger>
 </Syslog>
```

3.2.4 Test Scenario

The test harness is a simple program that constructs and sends an email to its SMTP server. The test is repeated 250 times to take a statistical average. The test harness is provided below:

```
import gov.ed.sfa.ita.email.SFASmtpClient;
import java.util.Date;
/*test harness for email framework.
 *The harness constructs and sends email to its smtp server.
 *The task is repeated for 250 times.
 */
public class emailTest {
      //the "to" field of the email message
      private static String to = "nullcline@yahoo.com";
      //the "from" field of the eamil message
      private static String from = "chi-yen.yang@accenture.com";
      //the "reply to" field of the email message
      private static String rpt = "nullcline32@hotmail.com";
      //the "subject" of the email message
      private static String subject = "test";
      /*constructor
       *constructs and sends email
       */
      public static void main (String args[]) {
            //sent date
            Date date;
            //loop counter
            int i;
            for(i = 0; i < 250; i++) {
                  date = new Date();
                  SFASmtpClient emailClient = new SFASmtpClient();
                  emailClient.setTextContent("This is a test");
                  emailClient.setFileAttachments("c:\\test.tst");
                  try {
                        emailClient.sendMessage(from, to, rpt,
subject);
                  }
                  catch(gov.ed.sfa.ita.exception.SFAException ex) {
                        System.out.println("Exception");
                  }
            }
      }
}
```

3.2.5 Analysis

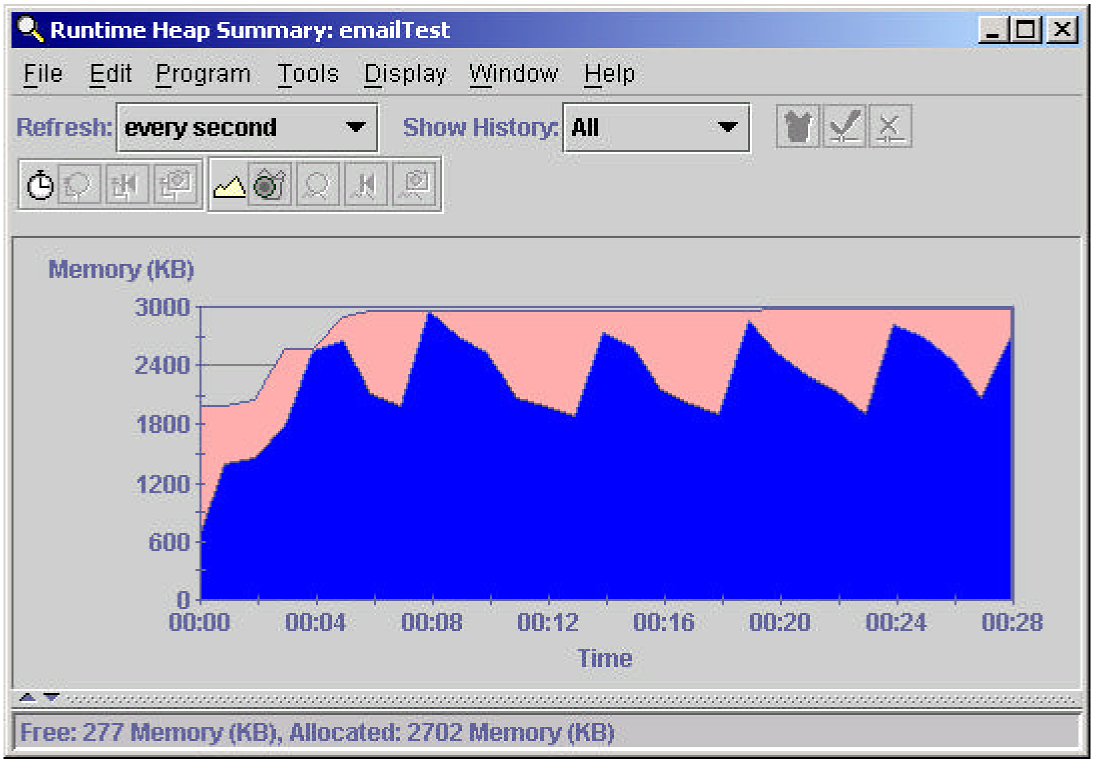
The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: Identifies any performance bottlenecks due to inefficient code algorithms.

3.2.5.1 Memory (Heap) Usage



Refresh: every second

Show History: All

Free: 277 Memory (KB), Allocated: 2702 Memory (KB)

JProbe's heap analysis tool shows the memory usage allocated by the JVM. The pink portion of the graph indicates the maximum heap allocated and the blue portion is the memory used at any given time. As one can see, there is a jump in memory allocation initially. The memory set at this stage is the default JVM memory allocation. As time passes, a jagged pattern develops. This type of graph indicates that there are a lot of short-lived objects used in the program that require constant garbage collection.

This particular pattern is expected since the harness runs repetitive tasks that generate short-lived objects. This is not an indication that the framework itself generates more than necessary short-lived objects.

3.2.5.2 Garbage Collection

Package	Class	GC'ed	Alive	Allocated At
java.lang	String	23,869	365	
	char[]	21,872	148	
java.lang	String	21,872	210	

Package	Class	GC'ed	Alive	Allocated At
	char[]	19,040	79	
javax.mail.internet	hdr	15,687	63	
javax.mail.internet	HeaderTokenizer$Token	12,250	0	
	char[]	8,825	31	
	char[]	8,035	199	
	Object[]	7,997	57	
	Object[]	5,531	94	

The garbage collections matrix shows the top 10 classes that are garbage collected. As shown in the matrix, String and char classes occupy most of the garbage collection process. Two classes from javax.mail.internet package are also placed in the top 10 list.

From JProbe's Heap analyzer, SFASmtpClient class has a change of 0. A "change" is the difference between the number of beginning and the number of end instances in a particular class. A change of 0 means all instances of a class are garbage collected and there are no loitering objects. Since SFASmtpClient is the class that is used to access the email framework, it can be concluded that there is no loitering objects produced in the framework.

3.2.5.3 Code Efficiency

Five performance matrices are collected for code efficiency analysis.

Top 10 methods with most calls:

Name	Package	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects
String.charAt(int)	java.lang	356,152	1,028 (4.8%)	1,028 (4.8%)	0 (0.0%)	0 (0.0%)
String.equalsIgnoreCase(String)	java.lang	184,535	711 (3.3%)	570 (2.7%)	0 (0.0%)	0 (0.0%)
String.indexOf(int, int)	java.lang	181,570	640 (3.0%)	640 (3.0%)	0 (0.0%)	0 (0.0%)
String.indexOf(int)	java.lang	179,863	1,705 (8.0%)	1,069 (5.0%)	0 (0.0%)	0 (0.0%)
BufferedInputStream.ensureOpen()	java.io	110,266	221 (1.0%)	221 (1.0%)	0 (0.0%)	0 (0.0%)
BufferedInputStream.read()	java.io	110,250	687 (3.2%)	402 (1.9%)	2 (0.0%)	0 (0.0%)
Vector.elementAt(int)	java.util	100,798	205 (1.0%)	205 (1.0%)	0 (0.0%)	0 (0.0%)
System.arraycopy(Object, int, Object, int, int)	java.lang	99,519	261 (1.2%)	261 (1.2%)	0 (0.0%)	0 (0.0%)
Vector$1.hasMoreElements()	java.util	83,500	143 (0.7%)	143 (0.7%)	0 (0.0%)	0 (0.0%)
Vector$1.nextElement()	java.util	79,750	137 (0.6%)	137 (0.6%)	0 (0.0%)	0 (0.0%)

As shown in the matrix, java.lang, java.io and java.util are most used in the framework.

Top 10 methods with most cumulative time:

Class	Package	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects
SFASmtpClient.sendMessage(String, String, String, String)	gov.ed.sfa.ita.email	250	21,297 (100.0%)	19 (0.1%)	280,843 (100.0%)	776 (0.3%)
SFASmtpClient.buildSession()	gov.ed.sfa.ita.email	250	18,744 (88.0%)	43 (0.2%)	243,915 (86.9%)	600 (0.2%)
Transport.send(Message)	javax.mail	250	14,322 (67.3%)	4 (0.0%)	176,874 (63.0%)	0 (0.0%)
Transport.send0(Message, Address[])	javax.mail	250	7,466 (35.1%)	34 (0.2%)	68,357 (24.3%)	2,003 (0.7%)
MimeMessage.saveChanges()	javax.mail.internet	250	6,188 (29.1%)	3 (0.0%)	103,517 (36.9%)	0 (0.0%)
MimeMessage.updateHeaders()	javax.mail.internet	250	6,186 (29.0%)	10 (0.0%)	103,517 (36.9%)	258 (0.1%)
MimeBodyPart.updateHeaders(MimePart)	javax.mail.internet	750	5,533 (26.0%)	72 (0.3%)	94,878 (33.8%)	1,004 (0.4%)
SMTPTransport.connect()	com.sun.mail.smtp	250	4,588 (21.5%)	2 (0.0%)	37,892 (13.5%)	0 (0.0%)
Service.connect()	javax.mail	250	4,586 (21.5%)	1 (0.0%)	37,892 (13.5%)	0 (0.0%)
Service.connect(String, String, String)	javax.mail	250	4,585 (21.5%)	1 (0.0%)	37,892 (13.5%)	0 (0.0%)

SFASmtpClient.sendMessage() method has the most cumulative time. This is expected because this method is the entry point for application code to JavaMail. SFASmtpClient.sendMessage() is responsible for building and sending email message and in the process calls on several other javax.mail.internet methods. Thus, aggregately, has the most time.

Top 10 methods with most method time:

Class	Package	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects
BufferedReader.<init>(Reader, int)	java.io	257	1,427 (6.7%)	1,426 (6.7%)	257 (0.1%)	257 (0.1%)
String.indexOf(int)	java.lang	179,863	1,705 (8.0%)	1,069 (5.0%)	0 (0.0%)	0 (0.0%)
String.charAt(int)	java.lang	356,152	1,028 (4.8%)	1,028 (4.8%)	0 (0.0%)	0 (0.0%)
String.indexOf(int, int)	java.lang	181,570	640 (3.0%)	640 (3.0%)	0 (0.0%)	0 (0.0%)
StringBuffer.expandCapacity(int)	java.lang	17,499	659 (3.1%)	614 (2.9%)	17,499 (6.2%)	17,499 (6.2%)
String.equalsIgnoreCase(String)	java.lang	184,535	711 (3.3%)	570 (2.7%)	0 (0.0%)	0 (0.0%)
HeaderTokenizer.getNext()	javax.mail.internet	16,000	1,596 (7.5%)	547 (2.6%)	33,250 (11.8%)	20,750 (7.4%)
String.<init>(char[], int, int)	java.lang	8,129	528 (2.5%)	505 (2.4%)	8,129 (2.9%)	8,129 (2.9%)
InternetHeaders.getHeader(String)	javax.mail.internet	3,750	1,205 (5.7%)	502 (2.4%)	12,750 (4.5%)	4,500 (1.6%)
InternetAddress.indexOfAny(String, String, int)	javax.mail.internet	5,250	1,491 (7.0%)	435 (2.0%)	0 (0.0%)	0 (0.0%)

The methods shown above are mostly called upon by the Email framework through JavaMail. These classes are essential in email constructing and sending processes.

Top 10 methods with most cumulative objects:

Name	Package	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects
SFASmtpClient.sendMessage(String, String, String, String)	gov.ed.sfa.ita.email	250	21,297 (100.0%)	19 (0.1%)	280,843 (100.0%)	776 (0.3%)
SFASmtpClient.buildSession()	gov.ed.sfa.ita.email	250	18,744 (88.0%)	43 (0.2%)	243,915 (86.9%)	600 (0.2%)
Transport.send(Message)	javax.mail	250	14,322 (67.3%)	4 (0.0%)	176,874 (63.0%)	0 (0.0%)
MimeMessage.updateHeaders()	javax.mail.internet	250	6,186 (29.0%)	10 (0.0%)	103,517 (36.9%)	258 (0.1%)
MimeMessage.saveChanges()	javax.mail.internet	250	6,188 (29.1%)	3 (0.0%)	103,517 (36.9%)	0 (0.0%)
MimeBodyPart.updateHeaders(MimePart)	javax.mail.internet	750	5,533 (26.0%)	72 (0.3%)	94,878 (33.8%)	1,004 (0.4%)
Transport.send0(Message, Address[])	javax.mail	250	7,466 (35.1%)	34 (0.2%)	68,357 (24.3%)	2,003 (0.7%)
ContentType.<init>(String)	javax.mail.internet	3,250	1,746 (8.2%)	73 (0.3%)	48,750 (17.4%)	6,500 (2.3%)
Service.connect(String, int, String, String)	javax.mail	250	4,584 (21.5%)	26 (0.1%)	37,892 (13.5%)	1,009 (0.4%)
SMTPTransport.connect()	com.sun.mail.smtp	250	4,588 (21.5%)	2 (0.0%)	37,892 (13.5%)	0 (0.0%)

As indicated in the matrix, SFASmtpClient.sendMessage() method has the most cumulative objects. The reason is the same as in cumulative method time. SFASmtpClient.sendMessage() initiates the calls to other methods and aggregately has the most objects.

Top 10 methods with most method objects:

Name	Package	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects
String.substring(int, int)	java.lang	26,888	473 (2.2%)	412 (1.9%)	24,738 (8.8%)	24,738 (8.8%)
StringBuffer.<init>(int)	java.lang	21,139	98 (0.5%)	98 (0.5%)	21,139 (7.5%)	21,139 (7.5%)
HeaderTokenizer.getNext()	javax.mail.internet	16,000	1,596 (7.5%)	547 (2.6%)	33,250 (11.8%)	20,750 (7.4%)
StringBuffer.toString()	java.lang	20,394	544 (2.6%)	394 (1.9%)	20,394 (7.3%)	20,394 (7.3%)
StringBuffer.expandCapacity(int)	java.lang	17,499	659 (3.1%)	614 (2.9%)	17,499 (6.2%)	17,499 (6.2%)
InternetHeaders.<init>()	javax.mail.internet	750	557 (2.6%)	349 (1.6%)	18,861 (6.7%)	16,542 (5.9%)
String.<init>(char[], int, int)	java.lang	8,129	528 (2.5%)	505 (2.4%)	8,129 (2.9%)	8,129 (2.9%)
Vector.<init>(int, int)	java.util	8,020	318 (1.5%)	274 (1.3%)	8,020 (2.9%)	8,020 (2.9%)
Object.clone()	java.lang	7,929	46 (0.2%)	46 (0.2%)	7,929 (2.8%)	7,929 (2.8%)
ParameterList.<init>(String)	javax.mail.internet	3,750	498 (2.3%)	57 (0.3%)	18,250 (6.5%)	7,500 (2.7%)

As shown in the matrix, objects are mostly coming from java.lang, java.util and javax.mail.internet packages.

3.2.6 General Performance Metrics

RCS Email framework is tested on Windows 2000 platform and JDK 1.3 running on PIII 800 MHz with 256M RAM machine. The test is run for an minute to gather the number of calls to SFASmtpClient.sendMessage() method. Two sets of numbers are collected: email with attachment (300K) and email without attachment.

Operation	Average Time	Calls Per Second
Email with attachment	0.092 s	10.8
Email without attachment	0.065 s	15.3

The five matrices collected above show that the calls to JavaMail methods within SFASmtpClient.sendMessage() method are necessary. The RCS email framework provides easy access to JavaMail for developers. Majority of the time is spent on JavaMail activities that are required for a proper-formed email message and there is a performance drop when attachment is included in an email message.

3.3 User Guide

3.3.1 Introduction

3.3.1.1 Purpose

This section provides a high-level summary, feature list, and usage of the Integrated Technical Architecture (ITA) standard Email Framework. The Email Framework is part of a suite of Reusable Common Services (RCS) provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects.

3.3.1.2 Intended audience

This section is intended for ITA and SFA application programmers who need to understand the Java Email framework in order to use this framework in their application.

3.3.1.3 Background

In the past SFA applications have used different methods to achieve Java Mail Messaging. They have used previous versions of the Java Mail API and also propriety designs that have included retrieving messages out of a oracle database and using Solaris shell scripts that use sendmail. Due to application support and availability issues, it's become very obvious that the ITA needs to specify a standard email API that is well documented and well tested.

3.3.1.4 Scope

This section covers only components that directly compose the Email framework. Consult the Sun Java website (http://www.javasoft.com) or applicable Java programming guides for more information on topics outside of the Email framework.

3.3.1.5 Assumptions

It is assumed that this framework will function in a J2EE application server environment. As the current production server for SFA is IBM WebSphere 3.5.3, the framework will be compiled using its required JDK version 1.2.2. It should also work with the current JavaServer Pages (1.1), Java Servlet (2.2), Java Messaging Service (1.0.1), and Java Database Connectivity (2.0) specifications for this server.

3.3.2 Description

3.3.2.1 Overview

Email functionality is an important system requirement in areas such as E commerce and customer care. In most cases, business applications use email to alert their customers of a security change or a News broadcast. The ITA Email framework is a wrapper class that simplifies the interface to Sun Microsystems's JavaMail. Sun Microsystems JavaMail API 1.2 provides a standard interface for Java Application programs to send emails to a SMTP Mail server. The JavaMail API is in turn built upon the Java Activation Framework (JAF), allowing complex message submitting and retrieving through different protocols. These API's provides a platform independent and protocol independent framework to build Java technology based mail and messaging applications.

The goal of the ITA Email Framework is to provide a simple yet robust interface to JavaMail 1.2 that could easily be utilized by any SFA development team that are building applications in Java that need to run in a Java Application Server. In the past SFA Application Operations Groups have reported limited success in tracing and debugging Email problems. It is very important that the ITA Email Framework be reliable and also very easy to debug if problems occur. To this end the ITA logging and Exception Handling Framework will be used to document any errors that the Email Framework encounters.

The Email framework will also standardize all SFA applications on a single interface to email. As specified before, many different implementations of email have occurred in previous projects and it is important that a supportable, maintainable standard email interface be used across many projects.

3.3.2.2 Features

The ITA Email Framework has a single class called SFASmtpClient. This bean provides simplified interfaces to the Sun Microsystems JavaMail API and provides public methods to create and manipulate email via Java programs.

In particular the SFASmtpClient bean provides the following features.

- Establish a real-time JavaMail Session with a SMTP Server
- Ability to set the subject, FromAddress, ToAddress, FileAttachment, ReplyToAddress, Sentdate and TextContent of current email.
- Ability to set multiple email addresses within the ToAddress, From Address and ReplytoAddresses
- Ability to get and set which SMTP Server the Email Framework will attach to.
- Ability to verify email parameters has meet minimum standards for delivery.

3.3.3 Installation

3.3.3.1 Software requirements

The Email Framework requires that a Simple Mail Transport Protocol Server (SMTP) is available so that emails generated by the client application may route mail to it. Currently ITA uses ***sendmail*** that is available on Sun Solaris or HP-UX as the standard SMTP server.

The Email framework uses JavaMail 1.2 to build and route email. The JavaMail API's are provided via the following Jar files. These jar files must be within the classpath.

Activation.jar
Mail.jar
MailApi.jar
rcs_email_v1.5.jar
Pop3.jar
Smtp.jar

The Email Framework uses the ITA Logging Framework, which requires the following Jar files to be within the classpath.

Jakarta-oro-2.0.1
Jdom-B6
Protomatter-1-1-5.jar
Utility.jar
Xerces.jar
Xml.jar
rcs_logging_v1.5.jar

The Email Framework uses the ITA Exception Handling Framework, which requires the following Jar files to be within the classpath.

rcs_exception_v1.5.jar

3.3.3.2 Usage

When using the Email Framework the Java developer would perform the following sequences to setup an email.

Task	Type of Variable or Method call
Specify a **from** Address variable and initialize it to a email address	String or IA*
Specify a **to** Address variable and initialize it to a email address(s)	String, String[] or IA[]*
Specify a **reply To Address** variable and initialize it to a email address(s)	String, String[] or IA[]*
Specify a **subject** variable and initialize it to the subject text	String
Build an instance of SFASmtpClient	SFASmtpClient()
Set the Text Content by calling the public method **setTextContent()**	SetTextContent(String text)

Task	Type of Variable or Method call

Set the fileAttachments (if needed) by calling the public method **setFileAttachments()**	SetFileAttachments(String fileloc)
	SetFileAttachments(String[filelocA, filelocB])

Send the Email by calling **sendMessage**() within a try loop catching for SFAException	SendMessage(from,to,reply,subj)

*IA represents object type **javax.mail.internet.InternetAddress**

Beware that the Email Framework requires that the From, To and ReplyTo arguments in ***sendMessage()*** have a valid address for it to successfully send an email. If these parameters are not set then a SFAException exception will be thrown. Another requirement is that all emails must follow the Email address specifications detailed in RFC822. If an email is found to be outside of the RFC822 specifications then an SFAException is thrown. (See Reference Section for link)

3.3.4 Configuration

Due to the fact that the Email Framework uses the SFA Logging and Exception handling framework then the needed logging and exception handling configuration files need to be in place to allow the Email Framework to function properly.

The logging framework uses a file called rcs.xml. This file is configured upon start of the WebSphere Application server and is available to all frameworks if the proper command line arguments are supplied to the Application Server. Each application shares the rcs.xml file with the common services used within that application. The location of the rcs.xml is specified via the command line argument Syslog.config.xml. Reference the ***ITA Reusable Common Services Logging*** User Guide for any questions.

The exception-handling framework uses an error message properties file that is located in the properties directory. The error messages appearing in the log file correspond to the code found in the properties file. Reference the ***ITA Reusable Common Services Exception Handling*** User Guide for any questions.

3.3.5 Usage Scenarios

Two examples are provided to illustrate the Email Framework.

The first example referenced in section 5.1 illustrates a Servlet example. The Servlet receives the Email addresses (toaddr, fromaddr, reply) dynamically through a web page. (Not shown) The servlet retrieves these addresses via the request object and the ***getParameter()*** public method. Once the addresses are known, a new Email object is built via the SFASmtpClient constructor. The text content of the Email is established via the SFASmtpClient public method ***setTextContent()***. If a file attachment is needed then a public method called ***setFileAttachments()*** is available to set a File Attachment location that the file is retrieved from. Finally the public method sendMessage is called to send the email. ***sendMessage()*** builds the session with javamail and the SMTP server, checks for email address syntax, and sends the email. If a send message exception or address exception occurs, ***sendMessage()*** will pass back a SFAException.

3.3.5.1 Servlet Example

```
import gov.ed.sfa.ita.email.*;
import java.io.*;
import javax.servlet.*;
public class EmailDriver extends
javax.servlet.http.HttpServlet {
public void doGet(javax.servlet.http.HttpServletRequest
request, javax.servlet.http.HttpServletResponse response)
throws javax.servlet.ServletException, java.io.IOException {
performTask(request, response);
}
public void doPost(javax.servlet.http.HttpServletRequest
request, javax.servlet.http.HttpServletResponse response)
throws javax.servlet.ServletException, java.io.IOException {
performTask(request, response);
}
public void
performTask(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response) {
PrintWriter out = null;
SFASmtpClient emailS=null;
Syslog.log(EmailDriver.class, "RCS_EMAIL",  "performTask():
Getting Parameters", null, Syslog.INFO);
//Get Parameters from a http session
String toaddr= request.getParameter("Toaddress");
String fromaddr = request.getParameter("Fromaddress");
String reply = request.getParameter("ReplyToAddress");
String subj = request.getParameter("Subject");
String message = request.getParameter("Message");
//Build Email  Session
emailS = new SFASmtpClient();

//SET TEXT CONTENT

emailS.setTextContent(message);
//Set File Attachment (If needed)
emailS.setFileAttachments("/tmp/Test_attach.doc");
try
{

//SEND THE EMAIL

emailS.sendMessage(fromaddr,toaddr,reply,subj);
out = new PrintWriter(response.getOutputStream());
out.println("<HTML><BODY>");
out.println("Email Sent!");
out.println("</BODY></HTML>");
out.close();
}
catch(SFAException s1)
{
// route to error page
}
}
}
```

The second example is a simple class example that builds the email within the main function and does not receive the Email addresses dynamically. The simple class example still shows building an Email instance, setting the different required fields (From address, To address and reply) and setting the File Attachment location (If needed). The public method ***sendMessage()*** is called to interface with javaMail and route the email. The try loop that surrounds the sendMessage() function catches SFAException's.

3.3.5.2 Simple class Example

```
import javax.mail.internet.*;
import java.util.*;
public class SMTPTester {
public SMTPTester() {
super();
}
public static void main(java.lang.String[] args) {
// First set up a 'From', 'To' and 'Reply To' address
variable
String sFrom = new String("root@vdc.com");
String sTo = new String("to@vdc.com");
String sRta = new String("root@vdc.com");

// SET UP A SUBJECT variable

String sSubj =
new String("Test for the String Interface in the Email
Framework");

//BUILD A EMAIL INSTANCE

SFASmtpClient emailS = new SFASmtpClient();

//SET THE EMAIL TEXT CONTENT

emailS.setTextContent("This is the test for the String
interface");

//SET AN ATTACHEDFILE IF NEEDED

emailS.setFileAttachments("c:\temp\file.doc");
try {

//SEND THE MESSAGE TO JAVAMAIL AND SIT BACK AND RELAX

emailS.sendMessage(sFrom, sTo, sRta, sSubj);
} catch (gov.ed.sfa.ita.exception.SFAException e2) {

//THE ONLY EXCEPTION  THE DEVELOPER HAS TO WORRY ABOUT IS
SFAEXCEPTION

System.out.println("You have caught an exception");
//route to error page
}
}
```

3.3.6 Sample Code

The ITA RCS Email Framework includes the two classes specified in section 5. The classes are SMTPTester and EmailDriver.

3.3.7 Resources

Java Activation Framework JavaDocs online

http://java.sun.com/products/javabeans/glasgow/javadocs

JavaMail 1.2 JavaDocs Online

http://www.javasoft.com/products/javamail/1.2/docs/javadocs/index.html

RFC-822 Reference

http://www.w3.org/Protocols/rfc822

4 RCS – Exception Handling Framework

The exception handling framework will provide a simple yet useful framework that can easily be utilized by any SFA development team building applications in Java, or more specifically WebSphere (although there is nothing in this package that ties it to WebSphere). The three things this framework is intended to provide is consistency in approach, standardization of error messages, and out-of-the-box integration with the logging framework.

Error handling is an important piece of any development effort, and a standardized framework should go a long way towards easing this coding burden on application programmers.

4.1 Testing Conditions & Results

4.1.1 Automated Testing

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	constructor with no arguments	testSFAException	testSFAException	SFAException	SFAException()	SFAException object is created	
2	constructor with a string argument	testSFAException	testSFAException	SFAException	SFAException (String message)	SFAException object is created, setting the message field to "this is a test message" and value is retrieved	
3	set and get of addlInfo field	testSFAException	testAddlInfo	SFAException	setAddlInfo getAddlInfo	the string: "this is addlInfo" is set and retrieved	
4	set and get of arguments field	testSFAException	testArguments	SFAException	setArguments getArguments	object array of 2 strings: "Argument One" and "Argument Two" is set and retrieved	
5	set and get of className field	testSFAException	testClassName	SFAException	setClassName getClassName	the string "className" is set and retrieved	
6	set and get of errorCode field	testSFAException	testErrorCode	SFAException	setErrorCode getErrorCode	the long integer 999 is set and retrieved	
7	set and get of message field	testSFAException	testMessage	SFAException	setMessage getMessage	the string "this is the error message text" is set and retrieved	
8	set and get of methodName field	testSFAException	testMethodName	SFAException	setMethodName getMethodName	the string "methodName" is set and retrieved	
9	set and get of origException field	testSFAException	testOrigException	SFAException	setOrigException getOrigException	an Exception object is set and retrieved	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
10	get reference to SFAExceptionFactory instance	testSFAExceptionFactory	testGetInstance	SFAExceptionFactory	getInstance	a reference to the SFAExceptionFactory is obtained	
11	get a SFAException from the SFAExceptionFactory	testSFAExceptionFactory	testCreateException	SFAExceptionFactory	createException	a new SFAException is obtained	
12	get the message for an SFAException from the SFAExceptionFactory	testSFAExceptionFactory	testGetMessageLookup	SFAExceptionFactory	getMessage	the correct string representing the error text for this exception is obtained	
13	get the default message for an SFAException from the SFAExceptionFactory	testSFAExceptionFactory	testGetMessageDefault	SFAExceptionFactory	getMessage	the default string for an SFA exception is obtained	
14	get the nested error for an SFAException where the nested errror is of type SFAException	testSFAExceptionFactory	testGetNestedErrorSFAException	SFAExceptionFactory	getNestedError	the nested exception for the SFAException is returned	
15	get the nested error for an SFAException where the nested errror is of type RemoteException	testSFAExceptionFactory	testGetNestedErrorRemoteException	SFAExceptionFactory	getNestedError	the nested exception for the RemoteException is returned	
16	get the nested error for an SFAException where the nested errror is of type InvocationTargetException	testSFAExceptionFactory	testGetNestedErrorInvocationTargetException	SFAExceptionFactory	getNestedError	the nested exception for the InvocationTargetException is returned	
17	get the nested error for an SFAException where the nested errror is of type ServerCloneException	testSFAExceptionFactory	testGetNestedErrorServerCloneException	SFAExceptionFactory	getNestedError	the nested exception for the ServerCloneException is returned	
18	get the nested error for an SFAException where the nested errror is of type ExceptionInInitializerError	testSFAExceptionFactory	testGetNestedErrorExceptionInInitializerError	SFAExceptionFactory	getNestedError	the nested exception for the ExceptionInInitializerError is returned	
19	get the nested error for an SFAException where the nested errror is of type SQLWarning	testSFAExceptionFactory	testGetNestedErrorSQLWarning	SFAExceptionFactory	getNestedError	the nested exception for the SQLWarning is returned	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
20	get the nested error for an SFAException where the nested errror is of type SQLException	testSFAExceptionFactory	testGetNestedErrorSQLException	SFAExceptionFactory	getNestedError	the nested exception for the SQLException is returned	
21	get the nested error for an SFAException where the nested errror is of type WriteAbortedException	testSFAExceptionFactory	testGetNestedErrorWriteAbortedException	SFAExceptionFactory	getNestedError	the nested exception for the WriteAbortedException is returned	
22	get the nested error for an SFAException where the nested errror is of type ServletException	testSFAExceptionFactory	testGetNestedErrorServletException	SFAExceptionFactory	getNestedError	the nested exception for the ServletException is returned	

4.2 Performance Analysis

4.2.1 Summary

This section provides a summary, configuration, data and analysis collected from the performance test of the SFA Exception Handling Reusable Common Service. The purpose of the test and the accompanying document is to identify performance issues and considerations while providing an overview of the architecture and use of the component.

The review of the ITA Exception Handling component found no performance issues with the design or the code. SFAException extends the capabilities of the base class java.lang.Exception considerably. It gives the application developer the added function of external logging, nested exceptions, and extended and customizable error codes. However, this functionality does come at a cost. The SFAException does not perform as fast as java.lang.Exception, but it is not meant to be a substitute, but an extension of the existing functionality.

The performance of some additional functions is based on the components that SFAException depends on for these tasks. The RCS Logging framework can be used to perform the logging of the extended information within an SFAException. The performance of that component is beyond the scope of this section and is included in a separate performance test report. Likewise, the additional information can be included with an SFAException. However, this proves to have only a small effect on the performance of the component because Java passes values by reference.

4.2.2 Test Environment

ITA Exception Handling was run through a performance testing harness to determine the relative resources required for the different parts of the exception-handling framework. These basic operations are the creation, initialization and throwing of the SFAException. The harness tested each of the above operations independently and as a whole. The test was run on a Compaq Desktop EN, 733 Mhz Pentium III, with JDK 1.2.2 on Windows NT 4.0.

4.2.3 Test Configuration

The configuration of the testing harness is straightforward. Two static variables that can be set at the command line control which sections of the code to run. This allows testing the two steps of the exception handling process that are supplied explicitly within SFAException, exception initialization and throwing. This allows us to identify bottlenecks and trouble spots. Creating the exception is taken as a given for this test. There is no possibility of initializing an Exception, much less throw one, without the SFAException object; therefore it must be a consistent portion of the scenario. Specifying an argument that is not create or throw can test for SFAException creation.

The test harness is as follows:

```
import gov.ed.sfa.ita.exception.*;
import java.io.*;
import java.util.*;

public class SFAExceptionMetrics
{
      static boolean doCreate = false;
      static boolean doThrow  = false;

        public SFAExceptionMetrics()  {
        }


      public static void main(String[] argv) {
               int i = 0;
            if (argv.length < 1) {
                  doCreate = true;
                  doThrow  = true;
               }
            else {
                  for (int i = 0; i < argv.length ; i++) {
                        String temp = argv[i];
                        if (temp.equalsIgnoreCase("create"))
                              doCreate = true;
                        if (temp.equalsIgnoreCase("throw"))
                              doThrow  = true;
                  }
               }
        }

        public static void metricsSFA() {
               Date dtStart = new Date();
               for(int i=0; i<10000; i++) {
                       try {
                            SFAExceptionFactory fac =
SFAExceptionFactory.getInstance();
                            SFAException ex = new
SFAException();
                            if (doCreate) ex =
                            fac.createException(
                                       SFAException.class,

SFAException.EMA_ADDRESS_EXCEPTION,
                                       null,
                                       null,
                                       "metricsSFA",
                                       "Thrown",
                                       null);
                            if (doThrow) throw ex;
                       }
                       catch (SFAException e) {
                       }
               }
            Date dtStop  = new Date();
            System.out.println("Trial Time (ms) : " +
                        (dtStop.getTime() - dtStart.getTime()));
        }
}
```

4.2.4 Test Scenario

The testing scenario was the creation, initialization, and throwing of a generic SFAException. There was no handling of the exception, nor was there extensive initialization. This test was designed to pinpoint performance issues within the SFAException and SFAExceptionFactory classes.

4.2.5 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: To identify any performance bottleneck due to inefficient code algorithms

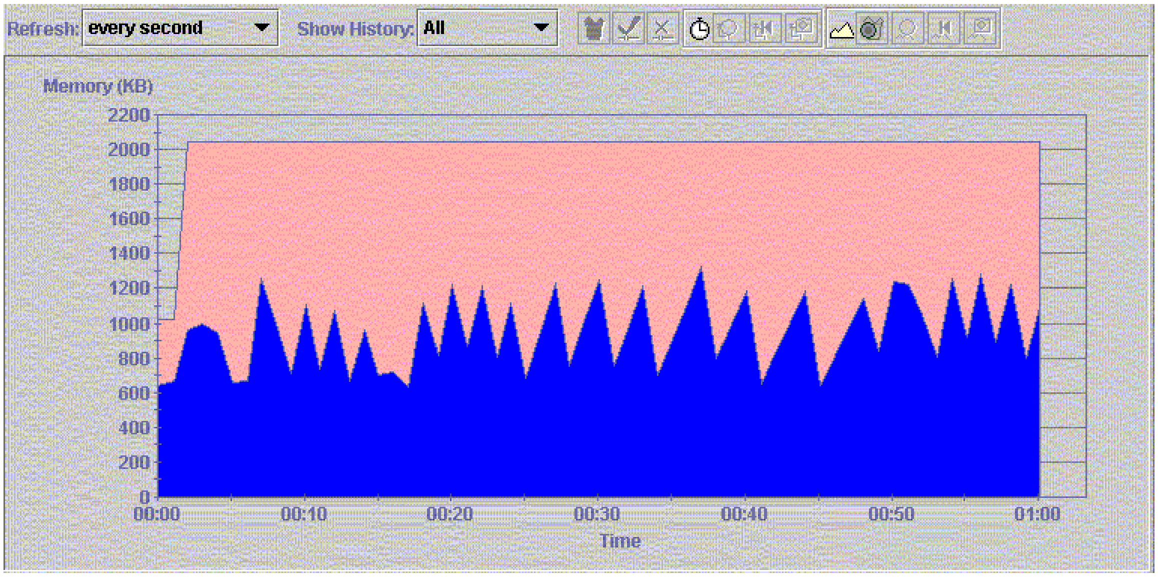
4.2.5.1 Memory (Heap) Usage

The performance test utilized JProbe Profiler's Memory Debugger to identify the sections of SFAException and SFAExceptionFactory that might be causing loitering objects. The following graphic is the Runtime Heap Summary, which is a pictorial rendering of the Java heap. Through analysis of the heap, we can find inconsistencies between predicted and expected memory use behavior and abnormal memory use behavior.

The Heap Usage Chart below plots the size of the Java heap at selected time intervals. The chart helps to visualize memory use in the Java heap. It displays the available size of the Java heap (in pink) and the used memory (in blue) over time.

The first part of the chart indicates a rounded hump. This rounded hump indicates the initialization of the test harness in memory. This is expected behavior, as the framework instantiates a set of objects to set up the test and then disposes of them. At this time the heap size (in pink) is also increased to handle the memory requirements of the test. This is normal behavior for the test harness and represents behavior for the harness, not for SFAException or SFAExceptionFactory.

The second part of the chart indicates a series of spikes. Steep spikes in the Heap Usage Chart represent temporary objects being allocated and garbage collected. If the level of the troughs becomes higher over time, then not all the temporary objects are garbage collected. As can be seen the troughs remain steady over time, and while multiple objects are being created and destroyed, there do not appear to be any lingering objects.



4.2.5.2 Garbage Collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage-collected objects accumulates. The list below is the final top ten, displaying cumulative objects created at the end of program execution.

Each row identifies the class by package name, if any, and class name. The next columns state, in order, the number of garbage collected objects (GC'ed column) for the class, the number of instances remaining in the heap (Alive column), and the method that allocated the instances of the class (Allocated At column). The same class can appear more than once because more than one method allocated instances of the class.

The chart below does not show any unexpected activity, or activity that would indicate a performance problem. Most of the objects created are strings, string buffers, or character arrays. These numbers are in line with the framework requirements and expected behavior as it formats a large number of SFAException messages and assigns them to the thrown SFAException.

Package	Class	GC'ed	Alive	Allocated At
	char[]	150,925	74	SFAExceptionJProbe.main
java.lang	StringBuffer	80,441	39	SFAExceptionJProbe.main
java.lang	String	70,546	37	SFAExceptionJProbe.main
	Object[]	30,511	15	Throwable.fillInStackTrace
	int[]	30,510	16	Throwable.fillInStackTrace
	Object[]	20,062	10	SFAExceptionJProbe.main
	int[]	20,062	10	SFAExceptionJProbe.main
java.util	Date	10,160	7	SFAExceptionJProbe.main
gov.ed.sfa.ita.exception	SFAException	10,159	5	SFAExceptionJProbe.main
gov.ed.sfa.ita.exception	SFAException	10,159	5	Class.newInstance()

4.2.5.3 Code Efficiency Metrics

There are nine efficiency metrics that can be collected in JProbe — five basic metrics and four compound metrics. The basic metrics include Number of Calls, Method Time, Cumulative Time, Method Object Count, and Cumulative Object Count. The compound metrics are averages per number of calls, including Average Method Time, Average Cumulative Time, Average Method Object Count, and Average Cumulative Object Count. Time is measured as elapsed time.

The following list defines the nine performance metrics:

- Number of Calls - The number of times the method was invoked.
- Method Time - The amount of time spent executing the method, excluding time spent in its descendants.
- Cumulative Time - The total amount of time spent executing the method, including time spent in its descendants but excluding time spent in recursive calls to descendants.
- Method Object Count - The number of objects created during the method's execution, excluding those created by its descendants.
- Cumulative Object Count - The total number of objects created during the method's execution, including those created by its descendants.
- Average Method Time - Method Time divided by Number of Calls.
- Average Cumulative Time - Cumulative Time divided by Number of Calls.
- Average Method Object - Count Method Object Count divided by Number of Calls.

The charts on the following pages serve to document the performance characteristics of the logging framework with several lists: methods with the most calls, methods with the most time spent in them, and methods with the most created objects. These measures are basic indicators of processing resource utilization. The lists can be reviewed for unexpected activity or optimization opportunities.

Methods with the most calls:

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
java.util	Date.getTime()	20,331	18	18	0	0	Date.java
java.util	Date.<init>()	10,166	53	53	0	0	Date.java
gov.ed.sfa.ita.exception	SFAException.<init>()	10,164	2,451	2,451	20,328	20,328	SFAException.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.createException (Class, long, Object[], Exception, String, String, String)	10,164	54,682	54,682	426,958	426,958	SFAExceptionFactory.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.getInstance()	10,164	399	399	2,076	2,076	SFAExceptionFactory.java
java.lang	ClassLoader.checkPackageAccess (Class, ProtectionDomain)	10	0	0	0	0	ClassLoader.java
java.lang	ClassLoader.loadClassInternal(String)	8	207	207	59	59	ClassLoader.java
	.Root.	1	202,151	1	485,997	0	
java.lang	Class.forName(String)	1	0	0	0	0	Class.java
java.lang	String.equalsIgnoreCase(String)	1	2	2	0	0	String.java
	.Signal dispatcher.	1	0	0	0	0	
	.Reference Handler.	1	69,753	69,753	0	0	
	.Finalizer.	1	69,752	69,752	0	0	

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
	.SymcJIT-LazyCompilation-PA.	1	0	0	0	0	
	.main.	1	62,646	2,430	485,997	16,142	
	.SymcJIT-LazyCompilation-0.	1	0	0	0	0	
	SFAExceptionJProbe.class$(String)	1	0	0	2	2	
	SFAExceptionJProbe.main(String[])	1	60,216	4	469,855	22	
	SFAExceptionJProbe.metricsSFA()	1	60,003	2,396	469,774	20,344	
java.util	Date.<clinit>()	1	2	2	66	66	Date.java
java.util	Date.<init>(long)	1	0	0	0	0	Date.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.<clinit>()	1	0	0	0	0	SFAExceptionFactory.java
	.Thread-0.	1	0	0	0	0	

Methods with the most total time:
(includes time spent in sub-methods)

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
	.Root.	1	202,151	1	485,997	0	
	.Reference Handler.	1	69,753	69,753	0	0	
	.Finalizer.	1	69,752	69,752	0	0	
	.main.	1	62,646	2,430	485,997	16,142	
	SFAExceptionJProbe.main(String[])	1	60,216	4	469,855	22	
	SFAExceptionJProbe.metricsSFA()	1	60,003	2,396	469,774	20,344	
gov.ed.sfa.ita.exception	SFAExceptionFactory.createException (Class, long, Object[], Exception, String, String, String)	10,164	54,682	54,682	426,958	426,958	SFAExceptionFactory.java
gov.ed.sfa.ita.exception	SFAException.<init>()	10,164	2,451	2,451	20,328	20,328	SFAException.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.getInstance()	10,164	399	399	2,076	2,076	SFAExceptionFactory.java
java.lang	ClassLoader.loadClassInternal(String)	8	207	207	59	59	ClassLoader.java
java.util	Date.<init>()	10,166	53	53	0	0	Date.java
java.util	Date.getTime()	20,331	18	18	0	0	Date.java
java.lang	String.equalsIgnoreCase(String)	1	2	2	0	0	String.java

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
java.util	Date.<clinit>()	1	2	2	66	66	Date.java
java.lang	ClassLoader.checkPackageAccess (Class, ProtectionDomain)	10	0	0	0	0	ClassLoader.java
java.lang	Class.forName(String)	1	0	0	0	0	Class.java
	.Signal dispatcher.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-PA.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-0.	1	0	0	0	0	
	SFAExceptionJProbe.class$(String)	1	0	0	2	2	
java.util	Date.<init>(long)	1	0	0	0	0	Date.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.<clinit>()	1	0	0	0	0	SFAExceptionFactory.java
	.Thread-0.	1	0	0	0	0	

Methods with the most method time:

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
	.Reference Handler.	1	69,753	69,753	0	0	
	.Finalizer.	1	69,752	69,752	0	0	
gov.ed.sfa.ita.exception	SFAExceptionFactory.createException (Class, long, Object[], Exception, String, String, String)	10,164	54,682	54,682	426,958	426,958	SFAExceptionFactory.java
gov.ed.sfa.ita.exception	SFAException.<init>()	10,164	2,451	2,451	20,328	20,328	SFAException.java
	.main.	1	62,646	2,430	485,997	16,142	
	SFAExceptionJProbe.metricsSFA()	1	60,003	2,396	469,774	20,344	
gov.ed.sfa.ita.exception	SFAExceptionFactory.getInstance()	10,164	399	399	2,076	2,076	SFAExceptionFactory.java
java.lang	ClassLoader.loadClassInternal(String)	8	207	207	59	59	ClassLoader.java
java.util	Date.<init>()	10,166	53	53	0	0	Date.java
java.util	Date.getTime()	20,331	18	18	0	0	Date.java
	SFAExceptionJProbe.main(String[])	1	60,216	4	469,855	22	
java.lang	String.equalsIgnoreCase(String)	1	2	2	0	0	String.java
java.util	Date.<clinit>()	1	2	2	66	66	Date.java

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
	.Root.	1	202,151	1	485,997	0	
java.lang	ClassLoader.checkPackageAccess (Class, ProtectionDomain)	10	0	0	0	0	ClassLoader.java
java.lang	Class.forName(String)	1	0	0	0	0	Class.java
	.Signal dispatcher.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-PA.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-0.	1	0	0	0	0	
	SFAExceptionJProbe.class$(String)	1	0	0	2	2	
java.util	Date.<init>(long)	1	0	0	0	0	Date.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.<clinit>()	1	0	0	0	0	SFAExceptionFactory.java
	.Thread-0.	1	0	0	0	0	

Methods with the most total objects:
(includes objects created in sub-methods)

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
	.main.	1	62,646	2,430	485,997	16,142	
	.Root.	1	202,151	1	485,997	0	
	SFAExceptionJProbe.main(String[])	1	60,216	4	469,855	22	
	SFAExceptionJProbe.metricsSFA()	1	60,003	2,396	469,774	20,344	
gov.ed.sfa.ita.exception	SFAExceptionFactory.createException (Class, long, Object[], Exception, String, String, String)	10,164	54,682	54,682	426,958	426,958	SFAExceptionFactory.java
gov.ed.sfa.ita.exception	SFAException.<init>()	10,164	2,451	2,451	20,328	20,328	SFAException.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.getInstance()	10,164	399	399	2,076	2,076	SFAExceptionFactory.java
java.util	Date.<clinit>()	1	2	2	66	66	Date.java
java.lang	ClassLoader.loadClassInternal(String)	8	207	207	59	59	ClassLoader.java
	SFAExceptionJProbe.class$(String)	1	0	0	2	2	
	.Reference Handler.	1	69,753	69,753	0	0	
	.Finalizer.	1	69,752	69,752	0	0	
java.util	Date.<init>()	10,166	53	53	0	0	Date.java

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
java.util	Date.getTime()	20,331	18	18	0	0	Date.java
java.lang	String.equalsIgnoreCase(String)	1	2	2	0	0	String.java
java.lang	ClassLoader.checkPackageAccess (Class, ProtectionDomain)	10	0	0	0	0	ClassLoader.java
java.lang	Class.forName(String)	1	0	0	0	0	Class.java
	.Signal dispatcher.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-PA.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-0.	1	0	0	0	0	
java.util	Date.<init>(long)	1	0	0	0	0	Date.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.<clinit>()	1	0	0	0	0	SFAExceptionFactory.java
	.Thread-0.	1	0	0	0	0	

Methods with the most methods objects:

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
gov.ed.sfa.ita.exception	SFAExceptionFactory.createException (Class, long, Object[], Exception, String, String, String)	10,164	54,682	54,682	426,958	426,958	SFAExceptionFactory.java
	SFAExceptionJProbe.metricsSFA()	1	60,003	2,396	469,774	20,344	
gov.ed.sfa.ita.exception	SFAException.<init>()	10,164	2,451	2,451	20,328	20,328	SFAException.java
	.main.	1	62,646	2,430	485,997	16,142	
gov.ed.sfa.ita.exception	SFAExceptionFactory.getInstance()	10,164	399	399	2,076	2,076	SFAExceptionFactory.java
java.util	Date.<clinit>()	1	2	2	66	66	Date.java
java.lang	ClassLoader.loadClassInternal(String)	8	207	207	59	59	ClassLoader.java
	SFAExceptionJProbe.main(String[])	1	60,216	4	469,855	22	
	SFAExceptionJProbe.class$(String)	1	0	0	2	2	
	.Root.	1	202,151	1	485,997	0	
	.Reference Handler.	1	69,753	69,753	0	0	
	.Finalizer.	1	69,752	69,752	0	0	
java.util	Date.<init>()	10,166	53	53	0	0	Date.java

Package	Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
java.util	Date.getTime()	20,331	18	18	0	0	Date.java
java.lang	String.equalsIgnoreCase(String)	1	2	2	0	0	String.java
java.lang	ClassLoader.checkPackageAccess (Class, ProtectionDomain)	10	0	0	0	0	ClassLoader.java
java.lang	Class.forName(String)	1	0	0	0	0	Class.java
	.Signal dispatcher.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-PA.	1	0	0	0	0	
	.SymcJIT-LazyCompilation-0.	1	0	0	0	0	
java.util	Date.<init>(long)	1	0	0	0	0	Date.java
gov.ed.sfa.ita.exception	SFAExceptionFactory.<clinit>()	1	0	0	0	0	SFAExceptionFactory.java
	.Thread-0.	1	0	0	0	0	

4.2.6 General Performance Metrics

ITA Exception Handling was run through a performance testing harness to determining the exception handling speed on a system. The harness tested each of the above operations independently and as a whole. Running a test on a Compaq Desktop EN, 733 Mhz Pentium III, with JDK 1.2.2 on Windows NT 4.0 yields the following performance numbers:

Operation	Average Time (ms)	Calls Per Second
Creation	0.00847	118063.7544
Initialization	0.93625	1068.090788
Throwing	0.05926	16874.78907
Total	1.00398	996.0357776

As the chart above indicates, the ITA Exception framework has strong performance characteristics and can function well under heavy loads. As this shows, the majority of the time spent within an SFAException is initializing the exception with the SFAExceptionFactory. This is to be expected, as it is the added functionality provided within SFAException and that is set using the SFAExceptionFactory that is the real advantage of the framework. This cost in processing time is acceptable for the tasks necessary to throw a custom exception.

4.2.7 Conclusion

SFAException extends the capabilities of the base class java.lang.Exception considerably. It gives the application developer the added function of external logging, nested exceptions, and extended and customizable error codes. However, this functionality does come at a cost. The SFAException does not perform as fast as java.lang.Exception, but it is not meant to be a substitute, but an extension of the existing functionality.

SFAException performs this task to a high standard. Upon analysis it showed no existing or potential bottlenecks within the code. There are no problems with the allocation and garbage collection of objects from the SFAException or SFAExceptionFactory, nor are there loitering objects that could signify a memory leak or misallocation.

The SFAException also performs well when run for speed and load. The numbers for the testing system show the capabilities of the SFAException under a load, to quickly create, initialize and throw a customized and informative exception. There are no performance deficiencies within the code or the design.

4.3 User Guide

4.3.1 Introduction

4.3.1.1 Purpose

This section provides a high-level summary and usage scenarios for the Integrated Technical Architecture (ITA) standard Exception Handling Framework. The exception handling framework is part of a suite of frameworks called Reusable Common Services (RCS), which are provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects.

4.3.1.2 Intended audience

This section is intended for ITA and SFA application programmers who need to understand the Exception Handling framework in order to use this framework in their application.

4.3.1.3 Background

The purpose of exception handling is to catch problems in a program that would otherwise lead to program error and failure conditions. An error in Java is an unrecoverable abnormal condition. For example, an error can occur when a networking connection is unexpectedly cut, or the JVM runs out of memory. The ITA Exception Handling framework exists to provide a mechanism by which client programmers may be able to trap exceptional conditions within their code. The benefits of having a framework to handle exceptions are to minimize duplicated effort and save time for programmers during both development and maintenance of the application. For more detail on the ITA Exception Handling framework please see the corresponding detailed design document.

4.3.1.4 Scope

This section covers only installation, features, and usage scenarios of the ITA Exception Handling framework. The Exception Handling framework is typically used with the ITA Logging framework – however, the Logging framework is not covered in this section. Consult the ITA Logging User Guide for more information on the Logging framework.

4.3.1.5 Assumptions

The Exception Handling framework has been designed and tested in a J2EE application server environment. Specifically, it was developed in the current production environment for SFA: IBM WebSphere 3.5, running on its required IBM JDK version 1.2.2. It should also work with the current JavaServer Pages (1.1), Java Servlet (2.2), Java Messaging Service (1.0.1), and Java Database Connectivity (2.0) specifications for this server. It was built and tested on Sun Solaris 2.6 operating system.

4.3.2 Description

4.3.2.1 Overview

The ITA Exception Handling framework is designed to provide a mechanism for trapping and dealing with any erroneous or unexpected actions that applications may encounter during runtime. The Exception Handling Framework also allows error message formatting standards to be developed and enforced to ensure proper information is gathered if a program failure occurs.

4.3.2.2 Features

The exception handling framework provided by Java is very generic. It gives very generic messages to the developer, which makes it difficult to find the actual cause of the exception. Because of this, a custom framework for exception handling can be of great benefit to a development effort. With a custom exception handling framework, individual exceptions can be identified and handled on a case-by-case basis.

The messages for exceptions are stored in a properties file, allowing for easily updated messages. The properties file is parameterized, however, so values can be inserted at run-time as needed. In this way the exception handling framework provides a error message handling mechanism that is at the same time standardized and dynamic.

4.3.2.3 Main Concepts

The ITA Exception Handling Framework enhances the ability of an application to define and trap errors that may arise as the application executes. The Exception Handling framework includes the following key components:

- SFAExceptionFactory
- SFAException

The SFAException class is the base exception, which can be customized on an application by application basis. It has set() and get() methods for each of its main properties, such as errorCode, methodName, className, and so forth.

The SFAExceptionFactory is the class used to create exceptions. It should work in any application without modifications. This class is designed to work as a singleton (i.e. there is only one instance of it per VM or at least per classloader). Therefore, a reference to the factory is obtained via the getInstance() method. It also contains some exception-centered utility methods, such as getNestedException(), which can be used to obtain the nested exception for all standard Java exception types.

4.3.3 Installation

4.3.3.1 Software requirements

The ITA Exception Handling Framework should work in any Java environment above JDK 1.2.2, including all Java application servers or standalone Java applications. However, it was designed and tested in SFA’s production server environment, IBM WebSphere 3.5.3 and on IBM’s JDK 1.2.2. It has no other software dependencies. However, if applications intend to log error messages using the ITA Logging framework, then the user guide for that framework should be consulted for the appropriate configuration and supporting libraries.

4.3.3.2 Installation procedures

The Exception Handling framework uses one jar file, rcs_exception_v1.5.jar, for operation. This jar file contains all necessary files for proper execution of the Exception Handling framework. The classpath of the application or web server should be updated to refer to this file. If the list of error codes is updated (see directly below, under ‘Configuration’) then the class SFAException will need to be recompiled and the jar file updated.

4.3.4 Configuration

The SFAException class distributed here for use with SFA development projects contains a set of generic error codes to serve as a starting point. Each project should spend time early on in the development cycle to define a list of error codes encompassing all of the potential error conditions that their application may encounter. These error codes should be updated in the set of error code constants defined in the SFAException class. The beginning of the error code section is listed below for reference:

```
public class SFAException extends Exception
{
      // List of private and protected class variables…
      // Generic Exceptions
      public static final long UNEXPECTED_EXCEPTION= 0;
      public static final long MISSING_PROPERTY= 1;
      public static final long BAD_PROPERTY_FORMAT= 2;
      public static final long ERROR_LOADING_PROPERTY_FILE= 3;
      public static final long NEW_INSTANCE_FAILED= 4;
      public static final long JNDI_INIT_ERROR= 5;
      public static final long INVALID_HOME_INTERFACE= 6;
```

With the exception codes defined, a list of corresponding exception messages should be created for the errorMessages_en_US.properties resource file. The message number in the resource file corresponds to the message constant defined in the customized SFAException handling class. A sample resource file is listed below:

```
# This file contains mapping information from error codes to error
messages
msg0="Unexpected exception caught"
msg1="Missing property {0}"
msg2="Bad property format {0} in property {1}"
```

The constant UNEXPECTED_EXCEPTION in the first list corresponds to the message "Unexpected exception caught" in the error messages file. This linkage is set by the number (0) that they share. Each entry in the messages file consists of a key – value pair where the key is in the form of:

'msg' + [the error code as defined in the SFAException class]

When a error code for an exception is set, the factory looks up the corresponding message by this key. Therefore, it is critical that these mappings are checked before the code and file is deployed so that appropriate messages are displayed / logged for the message codes.

Note that the message text for msg1 and msg2 contain integers surrounded by curly braces. For example:

"Missing property {0}"

These are parameterized fields which can be modified at run time. When a SFAException is created, values for these parameters can be set and inserted into the message text. This allows messages to be flexible and report run-time application status. An example of setting parameterized field follows in the usage section of this document.

4.3.5 Usage Scenarios

The following examples illustrate the main usage scenarios for the ITA Exception Handling framework.

4.3.5.1 Obtaining a reference to SFAExceptionFactory and generating a SFAException

```
      /** Get an instance of the factory. */
      gov.ed.sfa.ita.exception.SFAExceptionFactory sfaFac =
SFAExceptionFactory.getInstance();

      /** Create an array of Objects as message parameters.
Basically, these values will be passed to the message located in the
property file so that it can be substituted at run-time */
      String value1 = "<Dynamically substitute value 1>";
      String value2 = "<Dynamically substiture value 2>";
      Object[] arguments = {value1, value2};

      /** If this was being raised from a try-catch block then the
caught exception could be set as the original exception.  In this
case it is null. */
      Exception origException = null;

      /** Get the factory to create an exception, passing the
appropriate parameters.  */
      gov.ed.sfa.ita.exception.SFAException sfaExcep =
sfaFac.createException(SFAException.class,
```

4.3.5.2 Catching an error and throwing it, wrapped in an SFAException

```
try {
      // Some application code that may raise a SQLException, for
example
}
catch (SQLException e) {
      /** Get an instance of the factory. */
      gov.ed.sfa.ita.exception.SFAExceptionFactory sfaFac =
SFAExceptionFactory.getInstance();

      /** As this is being raised from a try-catch block then the
caught exception, e, is set as the original exception. */
      Exception origException = e;

      /** Get the factory to create an exception, passing the
appropriate parameters.  */
      gov.ed.sfa.ita.exception.SFAException sfaExcep =
sfaFac.createException(SFAException.class,

      gov.ed.sfa.ita.exception.SFAException.DEFINED_SQL_EXCEPTION_COD
      E,
      null,
      origException,
      "ExampleLoggingAndException",
      "main",
      null);

      /** Throw the new exception, with the nested SQLException. */
      throw sfaExcep;
}
```

4.3.5.3 Setting parameterized fields in the SFAException

```
      /** Get an instance of the factory. */
      gov.ed.sfa.ita.exception.SFAExceptionFactory sfaFac =
SFAExceptionFactory.getInstance();

      /** Use the factory to create an exception, passing the
appropriate parameters.  */
      gov.ed.sfa.ita.exception.SFAException sfaExcep =
sfaFac.createException(SFAException.class,
      gov.ed.sfa.ita.exception.SFAException.UNEXPECTED_EXCEPTION,
      null,
      null,
      "ExampleLoggingAndException",
      "main",
      null);

      /** If being used in an application, throw the exception. */
      throw sfaExcep;
```

4.3.6 Last Resort Handlers

Part of a complete exception handling framework is the accounting for a last-resort handling mechanism. A last-resort handler ensures that no application-level exception ever reaches the user without at least some processing. For example, there may be situations where the application has no alternatives but to report an error to the user and exit. In this case, a well written web application will at least reformat the error in a human-understandable format, provide links or phone numbers for help desk support to resolve the issue, and perhaps email or page system administrators with some captured state information.

In the IBM WebSphere environment, every Web application can specify a default error page that will be called in the event of exceptions or errors that are not handled directly in application code. The error page has access to the bean com.ibm.websphere.servlet.error.ServletErrorReport for accessing error information.   An example of the page is below:

Sample Last Resort Error Handling JavaServer Page: error_handler.jsp

```
<%@ page import="com.ibm.websphere.servlet.error.*" %>
<!--
   Simple error page for reporting application errors.  This error page
is called when a servlet throws an Exception, or by calling
response.sendError().  Error pages can use the request-scoped bean
named "ErrorReport" to get more information about the error.
--->
<jsp:useBean id="ErrorReport" scope="request"
class="com.ibm.websphere.servlet.error.ServletErrorReport"/>
<html>
<head><title>Error <%=ErrorReport.getErrorCode()%></title></head>
<body>

<% if (ErrorReport != null) { %>
<H1>Error : <%= ErrorReport.getErrorCode() %> </H1>
<H4>An error has occured while processing request: <%=
HttpUtils.getRequestURL(request) %></H4>
<B>TargetServletName: </B><%= ErrorReport.getTargetServletName() %><BR>
<B>Message: </B><%= ErrorReport.getMessage() %><BR>
<B>StackTrace: </B><%= ErrorReport.getStackTrace() %><BR>
<B>RootCause: </B><%= ErrorReport.getRootCause() %><BR>
<% } %>

</html>
```

This error page can be extended with more messages, application specific graphics, etc., to make it more appealing to a user who would encounter it.

Specifying the error page can be done via the administrative console. The setting can be made on the 'Advanced' tab under the
WebSphere Administrative Domain -> [host name] -> [Default (or Specified) Server] -> [Default (or Specified) Servlet Engine] -> [Web Application]
branch of the navigation pane.

In addition, every application JSP can specify its own error handling page through the <%@ page errorPage="error_handler.jsp"%> page directive. Then, when an unhandled error occurred in the application page, it would redirect to the error_handler.jsp page. The application level default error page would be the preferred mechanism, however, as it would ensure consistency in last-resort error handling for the application.

For more information on this bean and on default error pages in WebSphere, please consult the product documentation and JavaDocs (package `com.ibm.websphere.servlet.error`).

4.3.7 Resources

The following is a list of referenced and / or related publications.

The Exception Handling API JavaDocs

The ITA Homepage – http://www.ita.sfa.ed.gov

IBM WebSphere Infocenter -
http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/index.html

IBM WebSphere API documents -
http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/apidocs/index.html

5 RCS – Logging Framework

The logging framework enhances the current logging ability of the ITA Web application servers (specifically WebSphere), by allowing developers to dynamically set logging functionality without modifying tested source code. The logging framework also allows SFA to develop and enforce log formatting standards to ensure proper information is gathered if a failure occurs.

The framework is a simple and robust logging system that is not tied to any specific application server. It provides the following features:

- Simple logging API
- Background logging (*Configurable*)
- Multiple message severities
- Logs the name of the thread that issued the message (*Configurable*)
- Logs the name of the host that issued the message (*Configurable*)
- Arbitrary log channel names
- Pluggable log message listeners
- Pluggable log formatting modules for each listener
- Pluggable log policy modules for each listener
- Configuration can be modified on-the-fly while the system is running
- A running configuration can be written to XML for re-load later

The framework lets the programmer log messages easily through a simple API. During development and testing, the messages may be simply sent to the console where the application is running or to a single log file. When the system is moved into production, log messages can be split up by severity (for example, fatal messages may trigger the paging of an operations support resource) and log files may be rotated every night and archived. These kinds of configuration changes do not require changes to the code and can even be made while the system is running.

5.1 Testing Conditions & Results

5.1.1 Automated Testing

Logging to file

Important Preconditions:

- The method that loads the configuration file [Syslog.addLogging()] must be run in order to configure input/output settings.
- This method is inherently tested when testing all other methods, as it is not possible to use the logging framework without first calling this method.

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name [input]	Data File Name [output]
1	Check if a message is going to be logged.	TestLogging	testCanLog	Syslog	canLog	Return true if a message can be logged. Return false if a message cannot be logged	rcsFile.xml	rcslog.txt & console_output.txt
2	A message with INFO severity level is logged to the output file	TestLogging	testLog	Syslog	log	Log a message with an info severity level	rcsFile.xml	rcslog.txt & console_output.txt
3	A message with DEBUG severity level is logged to the output file	TestLogging	testLog	Syslog	log	Log a message with a debug severity level	rcsFile.xml	rcslog.txt & console_output.txt
4	A message with WARNING severity level is logged to the output file	TestLogging	testLog	Syslog	log	Log a message with a warning severity level	rcsFile.xml	rcslog.txt & console_output.txt
5	A message with ERROR severity level is logged to the output file	TestLogging	testlog	Syslog	log	Log a message with an error severity level	rcsFile.xml	rcslog.txt & console_output.txt
6	A message with FATAL severity level is logged to the output file	TestLogging	testlog	Syslog	log	Log a message with a fatal severity level	rcsFile.xml	rcslog.txt & console_output.txt

Logging to console

Important Preconditions:

- The method that loads the configuration file [Syslog.addLogging()] must be run in order to configure input/output settings.
- This method is inherently tested when testing all other methods, as it is not possible to use the logging framework without first calling this method.

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name [input]	Data File Name [output]
1	Check if a message is going to be logged.	TestLogging	testCanLog	Syslog	canLog	Return true if a message can be logged. Return false if a message cannot be logged	rcsConsole.xml	console_output.txt
2	A message with INFO severity level is logged on the console	TestLogging	testLog	Syslog	log	Log a message with an info severity level	rcsConsole.xml	console_output.txt
3	A message with DEBUG severity level is logged on the console	TestLogging	testLog	Syslog	log	Log a message with a debug severity level	rcsConsole.xml	console_output.txt
4	A message with WARNING severity level is logged on the console	TestLogging	testLog	Syslog	log	Log a message with a warning severity level	rcsConsole.xml	console_output.txt
5	A message with ERROR severity level is logged on the console	TestLogging	testlog	Syslog	log	Log a message with an error severity level	rcsConsole.xml	console_output.txt
6	A message with FATAL severity level is logged on the console	TestLogging	testlog	Syslog	log	Log a message with a fatal severity level	rcsConsole.xml	console_output.txt

Logging to file and console

Important preconditions:

- The method that loads the configuration file [Syslog.addLogging()] must be run in order to configure input/output settings.
- This method is inherently tested when testing all other methods, as it is not possible to use the logging framework without first calling this method.

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name [input]	Data File Name [output]
1	Check if a message is going to be logged.	TestLogging	testCanLog	Syslog	canLog	Return true if a message can be logged. Return false if a message cannot be logged	rcs.xml	rcslog.txt & console_output.txt

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name [input]	Data File Name [output]
2	A message with INFO severity level is logged on the console and to the output file	TestLogging	testLog	Syslog	log	Log a message with an info severity level	rcs.xml	rcslog.txt & console_output.txt
3	A message with DEBUG severity level is logged on the console and to the output file	TestLogging	testLog	Syslog	log	Log a message with a debug severity level	rcs.xml	rcslog.txt & console_output.txt
4	A message with WARNING severity level is logged on the console	TestLogging	testLog	Syslog	log	Log a message with a warning severity level	rcs.xml	rcslog.txt & console_output.txt
5	A message with ERROR severity level is logged on the console and to the output file	TestLogging	testlog	Syslog	log	Log a message with an error severity level	rcs.xml	rcslog.txt & console_output.txt
6	A message with FATAL severity level is logged on the console and to the output file	TestLogging	testlog	Syslog	log	Log a message with a fatal severity level	rcs.xml	rcslog.txt & console_output.txt

5.2 Performance Analysis

5.2.1 Summary

This section reviews the performance testing of the ITA Logging framework. Its purpose is to provide an architecture review of the framework and identify potential performance issues and performance considerations that an application development group should be aware of when using the framework. It consists of several areas of analysis:

- Architectural Overview
- Testing Environment and Configuration
- Results and Analysis:
  - Heap Usage
  - Garbage Collections
  - Code Efficiency
- General Metrics

A careful review of the ITA Logging framework revealed no performance deficiencies in its design or code. In fact, the review found that high performance was a key requirement in the design of the framework and the architecture of the framework serves to meet that requirement. Performance metrics further underline the capabilities of the framework.

Some areas of consideration were noted, however. These areas did not involve deficiencies in the framework itself. Instead it involved framework configurations and usages that would lead to slow performance. This behavior would be the result of formatting or logging messages in ways that are resource intensive or involve subsystem latencies. Examples of these usages would be formatting HTML, logging via e-mail, or logging via JMS. In these cases, the more sophisticated logging capabilities should be weighed against the added performance hit of their use.

5.2.2 Background

The main task that applications perform is usually not logging. Therefore, logging should not take up a significant amount of the system's resources nor interfere with its operation. The default implementations of loggers, log policies, and text formatters included with the ITA Logging framework have all been written with performance in mind.

ITA Logging framework implementations perform three operations when writing a message to a log file. These operations are described below:

1. Policy Check
   Syslog asks the policy if it should log a given message or not. This operation occurs most often. Assuming the use of the SimpleLogPolicy policy, the policy check consists of integer bitmask operations and, if channel sets are involved, a HashMap lookup. A policy check runs extremely fast in modern VMs -- on the order of millions of calls per second -- and inflicts almost no overhead even with extremely high message volumes. If that performance level isn't satisfactory, set the policy to accept the ALL_CHANNEL channel, and no HashMap lookup will occur. It is also common to use a static final boolean constant around debugging calls so that those calls can turn off completely at compile time, like this:

   ```
   public interface DebugFlag
   {
   public static final boolean DEBUG = true; // or false
   } ...
   if (DebugFlag.DEBUG) Syslog.debug(this, "Here is a debug message.");
   ```

2. Format Call
   The format call converts a log message to text. The default log formatter has been specifically optimized for speed. It uses StringBuffer objects and generally copies character arrays into those buffers for formatting, caching date format instances, and so forth.

3. Message write
   Actually writing the message to its final destination can be expensive. The file-based loggers all use buffered streams and have been extensively tested for speed. In addition, they are all necessarily thread-safe, so in the case of multithreaded applications, there can exist some unavoidable contention. Other loggers depend on other systems for their performance -- RemoteLog relies on the RMI implementation, and JMSLog depends on the JMS implementation -- so if the API providers are slow, the logger will be slow. The MailLog logger uses a background thread to connect to SMTP servers because this operation is generally quite slow.

5.2.3 Test Environment

The testing harness was run on a standard SFA developer workstation. The hardware consisted of a Compaq Deskpro with a single 600 MHz Pentium III processor and 512 MB of RAM. The machine ran

Windows NT 4.0 Service Pack 6. The Java environment was Sun's JDK 1.3. While tests were run, no other applications were loaded into memory, and the system was not interacted with. This was done in order to leave all resources available to the test harness, and eliminate the possibility of unexplained behavior in the tables of results.

5.2.4 Test Configuration

The ITA Logging framework was configured in a very standard manner, as it would be in actual usage. The configuration of the framework is done through its XML configuration file. The contents of the file as used are listed below. Key points of note are:

- The default mask is set to "DEBUG"
- The standard syslog FileLog is the logger class
- The logger 'listens' for events on all channels
- The standard syslog SimpleLogPolicy is the policy check class
- The standard syslog SimpleSyslogTextFormatter is the formatting class

```
<Syslog defaultMask="DEBUG">
 <Logger class="com.protomatter.syslog.FileLog" name="FileLogger">
  <Policy class="com.protomatter.syslog.SimpleLogPolicy">
   <channels>ALL_CHANNEL</channels>
   <logMask>INHERIT_MASK</logMask>
  </Policy>
  <Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
   <showChannel>false</showChannel>
   <showThreadName>false</showThreadName>
   <showHostName>false</showHostName>
   <dateFormat>HH:mm:ss MM/dd</dateFormat>
   <dateFormatCacheTime>1000</dateFormatCacheTime>
   <dateFormatTimeZone>America/New_York</dateFormatTimeZone>
  </Format>
  <fileName>output.txt</fileName>
  <append>true</append>
  <autoFlush>false</autoFlush>
 </Logger>
</Syslog>
```

The ITA Logging test harness has its own configuration file, which controls such things as the type of logging and the number of iterations to perform. The contents of this file are listed below. Key points of note are:

- The severity level is set to "ERROR"
- The number of policy check iterations is 100,000
- The number of formatting calls is 10,000
- The number of full logs (which includes the policy check and format every time) is 1,000

```
<PerformanceTest>
  <Message>

<loggerClassname>com.protomatter.syslog.PerformanceTest</loggerClass
name>
    <level>ERROR</level>
    <channel>DEFAULT_CHANNEL</channel>
    <message>This is the short message text, it's right here.</message>
    <detail>
    </detail>
  </Message>
  <PolicyTest>true</PolicyTest>
  <FormatTest>true</FormatTest>
  <DirectTest>true</DirectTest>
  <SyslogTest>true</SyslogTest>
  <NumThreads>5</NumThreads>
  <PolicyIterations>100000</PolicyIterations>
  <FormatIterations>10000</FormatIterations>
  <LogIterations>1000</LogIterations>
</PerformanceTest>
```

5.2.5 Test Scenarios

The logging performance test focused on one usage scenario for its analysis: writing a message to a local file-based log on the test machine. The usage scenario covers all major components of the logging framework: policy check, format message, and message write. The policy check was run 100,000 times, the message format was run 10,000 times, and the message write was run 1,000 times, as the performance test configuration file indicates.

The policy check used a SimpleLogPolicy class, which does basic checking against channels and severity levels. This policy check mirrors the one most applications would be expected to use.

The policy check did not use the PerClassPolicy class, which can filter out messages depending on the message origination. It would be expected that this type of policy check would incur more processing and therefore be slower than a basic policy check. It is also more complex to configure. Because of this it be used only in situations where there is a specific need for such granularity.

The message format used the SimpleSyslogTextFormatter, which is the most straightforward of all the message formatters. It is also the formatter most likely to be used in a logging installation. Other formatters, such as the HTMLSyslogTextFormatter, would be expected to be slower as they format the message in a more sophisticated manner. These loggers would be used only where the formatting needs would merit the additional processing.

The message log used the FileLog logger, which is also one of the most straightforward but most widely used loggers. This is the type of logger that would be used, for example, by a web server writing messages to a local log file. Most other file loggers, such as the TimeRolloverLog and LengthRolloverLog, should perform similarly. The one exception would be the OpenFileLog – as it opens and closes the file between each write, use of this logger is significantly more expensive.

There are many loggers available, such as the DatabaseLog, MailLog, and JMSLog, which behave quite differently from the file loggers. These loggers will vary significantly in there performance, as they are dependant on other subsystems for their ultimate performance. For example, the DatabaseLog would depend on the database software, hardware, and configuration in place. There are many performance-impacting variables to consider when implementing any of these other loggers.

5.2.6 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: To identify any performance bottleneck due to inefficient code algorithms

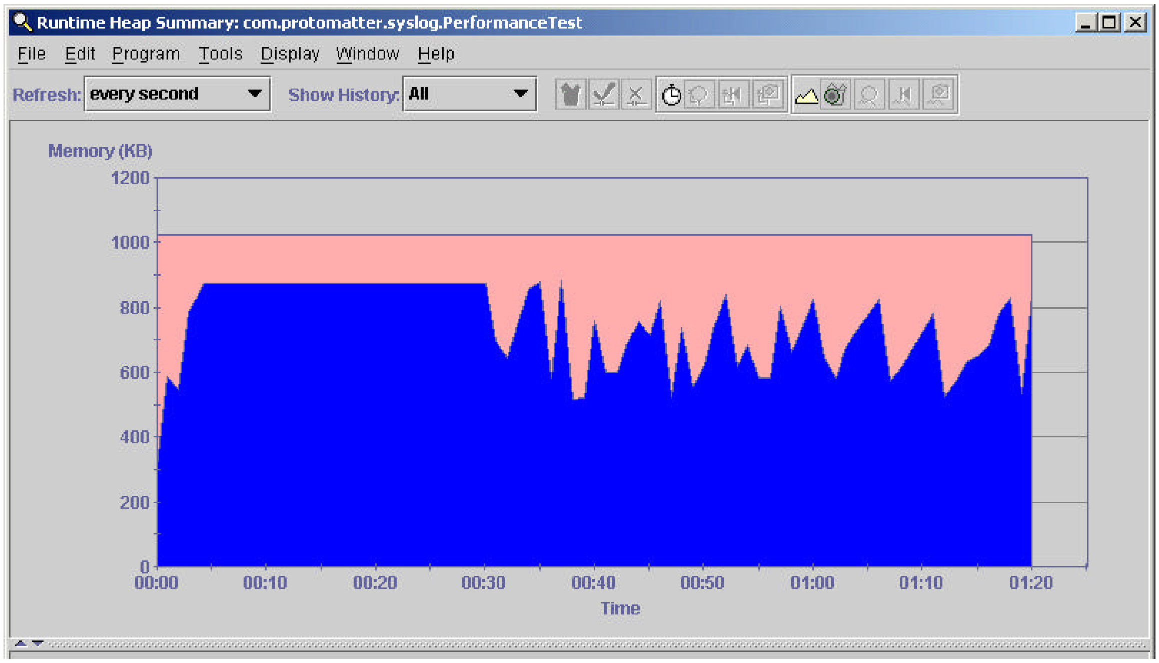
5.2.6.1 Memory (Heap) Usage

The performance test utilized JProbe Profiler's Memory Debugger to identify the parts of the logging framework that might be causing loitering objects. This was accomplished by analysis of the Java heap. The Runtime Heap Summary window can be used to view instance counts and information on allocating methods.

The Heap Usage Chart below plots the size of the Java heap at selected time intervals. The chart helps to visualize memory use in the Java heap. It displays the available size of the Java heap (in pink) and the used memory (in blue) over time.

The first part of the chart indicates a steady graph. A steady graph indicates that some core set of objects are remaining in memory. This is expected behavior, as the framework instantiates a set of core objects that remain in memory. This architecture helps performance as it minimzes recreation of commonly used objects.

The second part of the chart indicates a series of spikes. Steep spikes in the Heap Usage Chart represent temporary objects being allocated and garbage collected. If the level of the troughs become higher over time, then not all the temporary objects are garbage collected. As can be seen the troughs remain steady over time, and while multiple objects are being created and destroyed, there do not appear to be any lingering objects.



5.2.6.2 Garbage Collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage collected objects accumulates. The list below is the final top ten, displaying cumulative objects created at the end of program execution.

Each row identifies the class by package name, if any, and class name. The next columns state, in order, the number of garbage collected objects (GC'ed column) for the class, the number of instances remaining in the heap (Alive column), and the method that allocated the instances of the class (AllocatedAt column). The same class can appear more than once because more than one method allocated instances of the class.

The chart below does not show any unexpected activity, or activity that would indicate a performance problem. Most of the objects created are strings, string buffers, or character arrays. These numbers are in line with the framework requirements and expected behavior as it formats a large number of messages to write them to the file.

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	50,000	0	
java.lang	StringBuffer	9,611	389	
com.protomatter.syslog	SyslogMessage	4,611	389	
java.lang	StringBuffer	86	24	
java.lang	String	22	48	
	char[]	21	49	
com.protomatter.syslog	PerformanceTest$Test	15	5	
java.util	Date	12	1	
java.lang	String	10	1	

char[]	8	3

5.2.6.3 Code Efficiency

The charts on the following pages serve to document the performance characteristics of the logging framework with several lists: methods with the most calls, methods with the most time spent in them, and methods with the most created objects. These measures are basic indicators of processing resource utilization. The lists can be reviewed for unexpected activity or optimization opportunities.

Methods with the most calls:

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
SimpleLogPolicy.shouldLog(SyslogMessage)	com.protomatter.syslog	510,000	66,666	66,666	0	0	SimpleLogPolicy.java
StringBuffer.append(char[])	java.lang	360,000	10,894	10,894	0	0	StringBuffer.java
StringBuffer.append(char)	java.lang	300,000	7,659	7,659	0	0	StringBuffer.java
StringBuffer.append(String)	java.lang	60,275	2,258	2,258	68	68	StringBuffer.java
String.toCharArray()	java.lang	60,023	34,084	34,084	60,023	60,023	String.java
StringBuffer.append(Object)	java.lang	60,002	2,885	2,885	1	1	StringBuffer.java
String.toString()	java.lang	60,001	1,361	1,361	0	0	String.java
SimpleSyslogTextFormatter.formatLogEntry(StringBuffer, SyslogMessage)	com.protomatter.syslog	60,000	98,079	17,831	60,303	0	SimpleSyslogTextFormatter.java
SimpleSyslogTextFormatter.trimFromLastPeriod(StringBuffer, String, int)	com.protomatter.syslog	60,000	55,528	11,750	60,000	0	SimpleSyslogTextFormatter.java
StringBuffer.<init>(int)	java.lang	60,000	35,870	35,870	60,000	60,000	StringBuffer.java
SimpleSyslogTextFormatter.formatMessageDetail(StringBuffer, SyslogMessage)	com.protomatter.syslog	60,000	7,752	4,138	0	0	SimpleSyslogTextFormatter.java
StringBuffer.append(char[], int, int)	java.lang	60,000	2,035	2,035	0	0	StringBuffer.java

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
SimpleSyslogTextFormatter.formatDate(long)	com.protomatter.syslog	60,000	1,767	1,746	303	13	SimpleSyslogTextFormatter.java
SimpleSyslogTextFormatter.getStringForLevel(int)	com.protomatter.syslog	60,000	1,422	1,422	0	0	SimpleSyslogTextFormatter.java

Methods with the most total time (includes time spent in sub-methods):

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
.Root.		1	616,382	24	226,246	0	
PerformanceTest$TestThread.run()	com.protomatter.syslog	20	377,572	96,678	205,504	50,000	PerformanceTest.java
SimpleSyslogTextFormatter.formatLogEntry(StringBuffer, SyslogMessage)	com.protomatter.syslog	60,000	98,079	17,831	60,303	0	SimpleSyslogTextFormatter.java
FileLog.log(SyslogMessage)	com.protomatter.syslog	10,000	84,044	1,979	40,411	10,002	FileLog.java
.main.		1	80,594	1,013	20,738	1,438	
PerformanceTest.main(String[])	com.protomatter.syslog	1	79,574	23	19,217	279	PerformanceTest.java

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
.Reference Handler.		1	79,095	79,095	0	0	
.Finalizer.		1	79,094	79,094	0	0	
Thread.join()	java.lang	20	76,799	76,799	0	0	Thread.java
Writer.write(String)	java.io	10,001	72,030	72,030	120	120	Writer.java

Methods with the most time spent only in that method (not including sub-methods):

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
PerformanceTest$TestThread.run()	com.protomatter.syslog	20	377,572	96,678	205,504	50,000	PerformanceTest.java
.Reference Handler.		1	79,095	79,095	0	0	
.Finalizer.		1	79,094	79,094	0	0	
Thread.join()	java.lang	20	76,799	76,799	0	0	Thread.java
Writer.write(String)	java.io	10,001	72,030	72,030	120	120	Writer.java
SimpleLogPolicy.shouldLog(SyslogMessage)	com.protomatter.syslog	510,000	66,666	66,666	0	0	SimpleLogPolicy.java

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
StringBuffer.<init>(int)	java.lang	60,000	35,870	35,870	60,000	60,000	StringBuffer.java
String.toCharArray()	java.lang	60,023	34,084	34,084	60,023	60,023	String.java
SimpleSyslogTextFormatter.formatLogEntry(StringBuffer, SyslogMessage)	com.protomatter.syslog	60,000	98,079	17,831	60,303	0	SimpleSyslogTextFormatter.java
SimpleSyslogTextFormatter.trimFromLastPeriod(StringBuffer, String, int)	com.protomatter.syslog	60,000	55,528	11,750	60,000	0	SimpleSyslogTextFormatter.java
StringBuffer.append(char[])	java.lang	360,000	10,894	10,894	0	0	StringBuffer.java

Methods with the most total objects (includes objects created in sub-methods):

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
.Root.		1	616,382	24	226,246	0	
PerformanceTest$TestThread.run()	com.protomatter.syslog	20	377,572	96,678	205,504	50,000	PerformanceTest.java
SimpleSyslogTextFormatter.formatLogEntry(StringBuffer,	com.protomatter.syslog	60,000	98,079	17,831	60,303	0	SimpleSyslogTextFormatter.java

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
SyslogMessage)							
String.toCharArray()	java.lang	60,023	34,084	34,084	60,023	60,023	String.java
StringBuffer.<init>(int)	java.lang	60,000	35,870	35,870	60,000	60,000	StringBuffer.java
SimpleSyslogTextFormatter.trimFromLastPeriod(StringBuffer, String, int)	com.protomatter.syslog	60,000	55,528	11,750	60,000	0	SimpleSyslogTextFormatter.java
FileLog.log(SyslogMessage)	com.protomatter.syslog	10,000	84,044	1,979	40,411	10,002	FileLog.java
Syslog.log(InetAddress, Object, Object, Object, Object, int)	com.protomatter.syslog	5,000	46,671	131	35,390	0	Syslog.java
Syslog.log(Object, Object, Object, int)	com.protomatter.syslog	5,000	46,738	38	35,390	0	Syslog.java
Syslog.log(Object, Object, Object, Object, int)	com.protomatter.syslog	5,000	46,700	29	35,390	0	Syslog.java

Methods with the most objects created only in that method (not including sub-methods):

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
String.toCharArray()	java.lang	60,023	34,084	34,084	60,023	60,023	String.java
StringBuffer.<init>(int)	java.lang	60,000	35,870	35,870	60,000	60,000	StringBuffer.java
PerformanceTest$TestThread.run()	com.protomatter.syslog	20	377,572	96,678	205,504	50,000	PerformanceTest.java
StringBuffer.toString()	java.lang	10,110	1,928	1,928	10,110	10,110	StringBuffer.java
FileLog.log(SyslogMessage)	com.protomatter.syslog	10,000	84,044	1,979	40,411	10,002	FileLog.java
Thread.getName()	java.lang	5,000	2,639	2,639	10,000	10,000	Thread.java
SAXBuilder.build(File)	org.jdom.input	2	1,670	1,670	9,631	9,631	SAXBuilder.java
Syslog.log(InetAddress, Object, Object, Object, Object, int, Thread, String, long)	com.protomatter.syslog	5,000	43,812	1,037	25,390	5,002	Syslog.java
DecimalFormat.<init>(String)	java.text	4	130	130	2,252	2,252	DecimalFormat.java
String.toCharArray()	java.lang	60,023	34,084	34,084	60,023	60,023	String.java
StringBuffer.<init>(int)	java.lang	60,000	35,870	35,870	60,000	60,000	StringBuffer.java
PerformanceTest$TestThread.run()	com.protomatter.syslog	20	377,572	96,678	205,504	50,000	PerformanceTest.java

Name	Package	Calls	Total Time	Method Time	Total Objects	Method Objects	Source
StringBuffer.toString()	java.lang	10,110	1,928	1,928	10,110	10,110	StringBuffer.java
FileLog.log(SyslogMessage)	com.protomatter.syslog	10,000	84,044	1,979	40,411	10,002	FileLog.java

5.2.7 General Performance Metrics

ITA Logging framework was run through a performance testing harness to determine system-level performance metrics for the logger. The harness tested each major operation independently and the system as a whole, in single-threaded and multithreaded environments. That proves extremely useful when developing custom loggers, policies, and formatters. The test was run, multithreaded, on a Compaq Deskpro with an Intel 600 MHz Pentium III processor, running WindowsNT 4.0 and JDK 1.3. the test yielded the following performance numbers:

Operation	Average Time	Calls Per Second
Policy check (no channel check)	0.1 µs	8.2 million
Policy check (with channel check)	1.0 µs	982,000
Message format	11.9 µs	90,900
Aggregate Syslog.log() call, including file write	46.7 µs	21,400

ITA Logging performance statistics from Intel desktop test machine

As the chart above indicates, the ITA logging framework has strong performance characteristics and is capable of performing thousands of logging operations a second, which should meet the needs of most applications. Its tiered architecture of policy checkers, formatters, and loggers ensures that the only processing that occurs is what is absolutely necessary for an operation.

5.3 User Guide

5.3.1 Introduction

5.3.1.1 Purpose

This section provides a high-level summary, feature list, and usage of the Integrated Technical Architecture (ITA) standard Java Logging Framework. The logging framework is part of a suite of frameworks provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects.

5.3.1.2 Intended audience

This section is intended for ITA and SFA application programmers who need to understand the ITA Logging framework in order to use this framework in their application.

5.3.1.3 Background

The ITA custom logging feature set is implemented using an SFA-customized version of the Protomatter logging toolkit called Syslog. This toolkit is provided free for distribution through the Open Source

Software group, is licensed under the GNU Library General Public License Version 2, and is free for both commercial and non-commercial use. Specific terms of the license are available at http://www.gnu.org/copyleft/lgpl.html. The code has been modified from its initial state and the additional information is provided in this document to aid developers in using this framework for SFA projects.

5.3.1.4 Scope

This section covers installation, configuration, and features of the ITA logging framework. While the logging framework is usually used with the exception handling framework, the exception handling framework is not covered in this section. Consult the exception handling User Guide for more information on this framework.

5.3.1.5 Assumptions

It is assumed that the logging framework will function in a J2EE application server environment. As the current production server for SFA is IBM's WebSphere 3.5, the framework will be compiled using its required JDK version 1.2.2. It should also work with the current JavaServer Pages (1.1), Java Servlet (2.2), Java Messaging Service (1.0.1), and Java Database Connectivity (2.0) specifications for this server. It will be built and tested on Windows NT 4.0 and Sun Solaris 2.6 operating systems.

5.3.2 Description

5.3.2.1 Overview

The ITA Logging framework enhances the current logging ability of the ITA Web Application servers (specifically WebSphere), by allowing programmers to dynamically set logging and tracing functionality without modifying tested source code. The Logging Framework also allows log formatting standards to be developed and enforced to ensure proper information is gathered if a failure occurs.

5.3.2.2 Features

The ITA logging framework is not tied to any specific application server. The ITA logging framework provides the following features:

- Simple logging API
- Background logging (*Configurable*)
- Multiple message severities
- Logs the name of the thread that issued the message (*Configurable*)
- Logs the name of the host that issued the message (*Configurable*)
- Arbitrary log channel names
- Configuration can be modified on-the-fly while the system is running
- A running configuration can be written to XML for re-load later

5.3.2.3 Main Concepts

The Logging framework allows the programmer to log messages easily through a simple API. The Logging framework can be used as a debugging tool during development and as a troubleshooting tool when the application goes into production. During development and testing, the messages may be simply sent to the console where the application is running or to a single log file. When the system is moved into production, log messages can be split up by severity levels (Fatal messages may result in someone being paged, for instance) and log files may be rotated every night and archived. These kinds of configuration changes do not require changes to the code and can be made while the system is running. Detailed information on the logging framework can be found on the Logging design document.

5.3.3 Installation

5.3.3.1 Software requirements

The logging framework is J2EE compliant. The framework requires JDK 1.2 (recommended). It also works with JDK 1.3.

Operating System: The ITA team will test the logging framework in the following operating Systems: Windows NT 4.0 and Solaris 2.6

Application Server: The ITA team will test the logging framework in the WebSphere Application Server 3.5. There is not anything in the logging package that will tie the framework to a particular application server.

ITA Logging Package: The ITA logging package should include the following jar files:

- protomatter-1_1_5.jar
- jakarta-oro-2.0.1.jar
- jdom-B6.jar
- xerces.jar
- utility.jar
- xml.jar
- rcs_logging_v1.5.jar

5.3.3.2 Installation procedures

Copy all of the above files in a directory (eg /www/dev/rcs/jars).

5.3.4 Configuration

5.3.4.1 Add the Jar files on the classpath

The logging package needs to be added in WebSphere classpath. The following steps show how to add the classpath on WebSphere. Bring up the WebSphere admin console and select your application server on the console. Stop your application server Click on the 'General' Tab and add the following line in the Command Line Arguments**- classpath /www/dev/rcs/jars/.** Restart your application server.

5.3.4.2 Add the StartupRcs.jar on the classpath

Due to the fact that multiple applications can use the same ITA Reusable Common Services (RCS), it is beneficial to configure and launch a startup class that configures and starts any ITA RCS Services within an Application Server. This is accomplished by using WebSphere's ServiceInitializer interface. By specifying the name of the Startup Class as part of the ServiceInitializer command line argument for the Application Server, WebSphere will run the class as the last action it does in an Application Server startup or shutdown. An example is:

```
-Dcom.ibm.ejs.sm.server.ServiceInitializer=<class>[,<class>]...
```

where each <*class*> is one of the startup classes.

An example at SFA of this is the ITA Logging service. The logging service actually configures itself to the parameters specified within an XML configuration file (rcs.xml). The ITA RCS Startup class reads the XML file and configures the logging service upon startup of the Application Server. The StarupRcs class code can be found on the 'Sample code' section of this document.

To enable the RCS startup class the following steps need to be taken.

1. The StartupRcs.jar file needs to be placed within the classpath of the Application Server.. Bring the WebSphere Admin console up and select your application server. Stop your application server. Click on the 'General' Tab and add the path where StartupRcs.jar is located in the Command Line Arguments. Restart your application server

2. Add the following line in the Command Line Argument

   -**Dcom.ibm.ejs.sm.server.ServiceInitializer=gov.ed.sfa.ita.common.StartupRcs**

5.3.4.3 XML configuration file

The logging framework is configured by the xml configuration file. The xml file, *rcs.xml*, configures the loggers. The developer can configure the location of logs (file, console etc.), channel, thread, and date via the logging configuration file. Below is a sample of a logging configuration file. Details about the logging configuration file can be found at the following website:

http://protomatter.sourceforge.net/1.1.5/javadoc/com/protomatter/syslog/syslog-whitepaper.html

The developer needs to set up a variable for the xml configuration file on the WebSphere's Admin console. Add the following line in the Command Line Argument

-D Syslog.config.xml=<location of the logging configuration file eg. C:\rcs.xml>

The following is a sample configuration file (rcs.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<Syslog defaultMask="INFO" backgroundLogging="false">
<Logger class="com.protomatter.syslog.FileLog" name="rcs_sample">
 <fileName>d:\rcs_sample.log</fileName>
 <autoFlush>true</autoFlush>
  <stream>System.out</stream>
  <Policy class="com.protomatter.syslog.SimpleLogPolicy">
       <channels>ALL_CHANNEL</channels>
       <logMask>INHERIT_MASK</logMask>
   </Policy>
   <Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
       <showChannel>false</showChannel>
       <showThreadName>false</showThreadName>
       <showHostName>false</showHostName>
       <dateFormat>HH:mm:ss MM/dd</dateFormat>
       <dateFormatCacheTime>1000</dateFormatCacheTime>
       <dateFormatTimeZone>America/New_York</dateFormatTimeZone>
   </Format>
</Logger>
</Syslog>
```

The following sample logs the message to a console.

```
<?xml version="1.0" encoding="UTF-8"?>
<Syslog defaultMask="INFO" backgroundLogging="false">
<Logger class=”com.protomatter.syslog.SimpleSyslogTextFormatter”>
 <autoFlush>true</autoFlush>
  <stream>System.out</stream>
  <Policy class="com.protomatter.syslog.SimpleLogPolicy">
       <channels>ALL_CHANNEL</channels>
       <logMask>INHERIT_MASK</logMask>
   </Policy>
   <Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
       <showChannel>false</showChannel>
       <showThreadName>false</showThreadName>
       <showHostName>false</showHostName>
       <dateFormat>HH:mm:ss MM/dd</dateFormat>
       <dateFormatCacheTime>1000</dateFormatCacheTime>
       <dateFormatTimeZone>America/New_York</dateFormatTimeZone>
   </Format>
</Logger>
</Syslog>
```

5.3.5 Usage Scenarios

5.3.5.1 Using Logging Framework to a log simple text message

First, the developers need to import the ITA logging package in their application code (i.e. import gov.sfa.ed.ita.logging.*). The developers need to call the log method of the Syslog class to do all their logging. The log method has the following format.

```
Syslog.log(LogTest.class, LOG_CHANNEL, “This is a test message” , null , Syslog.INFO);
```

where LogTest.class is the name of the class where log method is called, LOG_CHANNEL is the name of the channel, “This is test message” – takes String or Object – used to log the detailed message, takes an object but here we have a null value , Syslog.INFO is the severity level of the log message.

Output of the above message will be as follows:

```
13:36:07 08/22 [INFO] LogTest   This is a test message
```

5.3.5.2 Using Logging Framework to log a message with different levels

The developers need to import the ITA logging package in their application code (i.e. import gov.sfa.ed.ita.logging.*). The developers need to call the log method of the Syslog class to do all their logging. The log method has the following format.

```
Object m_obj = new Object();

Syslog.log(LogTest.class, LOG_CHANNEL, “This is an informaiton test message” , Obj , Syslog.INFO);

Syslog.log(LogTest.class, LOG_CHANNEL, “This is a debug test message” , Obj , Syslog.DEBUG);

Syslog.log(LogTest.class, LOG_CHANNEL, “This is a warning test message” , Obj , Syslog.WARNING);

Syslog.log(LogTest.class, LOG_CHANNEL, “This is an error  test message” , Obj , Syslog.ERROR);

Syslog.log(LogTest.class, LOG_CHANNEL, “This is a fatal test message” , Obj , Syslog.FATAL);
```

where LogTest.class is the name of the class where log method is called, LOG_CHANNEL is the name of the channel. “This is a warning test message” – takes String or Object – used to log the detailed message Obj is an object, Syslog.INFO, Syslog.DEBUG, Syslog.WARNING, Syslog.ERROR, and Syslog.FATAL are the severity level of the log message.

Output of the above message will be as follows:

```
13:36:07 08/22 [INFO] LogTest   This is an information test message java.lang.Object 13:36:07 08/22
[DBUG] LogTest   This is a debug test message  java.lang.Object  13:36:07 08/22 [WARN] LogTest   This is
a warning test message  java.lang.Object 13:36:07 08/22 [EROR] LogTest   This is an error test message
java.lang.Object 13:36:07 08/22 [FTAL] LogTest.  This is a fatal test message java.lang.Object
```

5.3.6 Sample Code

The following sample code shows the different ways to use the ITA logging framework.

```
package gov.ed.sfa.ita.logging.examples;

import java.io.*;

import gov.ed.sfa.ita.logging.Syslog;

/**
 * LogTest:  This class is used to test the ITA Syslog methods.
 * Creation date: (8/10/01 3:25:49 PM)
 * @author: Roshani Bhatt
 */
public class LogTest {
/**
 * LogTest constructor
 */
public LogTest() {
      super();
}
/**
 * This method test all the ITA syslog class methods.
 * Creation date: (8/3/01 3:44:50 PM)
 * @param args java.lang.String[]
 */
 public static void main(String[] args)
 {
    Object m_obj = new Object();
    //testing canLog method
    boolean b1 = Syslog.canLog(Syslog.DEBUG);
    System.out.println("CanLog debug test produces " + b1);
    boolean b2 = Syslog.canLog(Syslog.INFO);
    System.out.println("CanLog info test produces " + b2);

     //Testing log method
     Syslog.log(LogTest.class, null, null, null, Syslog.INFO);
     Syslog.log(LogTest.class, "RCS_Logging", null, null,
Syslog.INFO);
     Syslog.log(LogTest.class, "RCS_Logging", "This is an INFO
message", null, Syslog.INFO);
    Syslog.log(LogTest.class, "RCS_Logging", "This is an INFO
message", m_obj, Syslog.INFO);
    Syslog.log(LogTest.class, "RCS_Logging", "This is a WARNING
message", m_obj,Syslog.WARNING);
     Syslog.log(LogTest.class, "RCS_Logging", "This is an DEBUG
message", m_obj,Syslog.DEBUG);
    Syslog.log(LogTest.class, "RCS_Logging", "This is an ERROR
message", m_obj,Syslog.ERROR);
     Syslog.log(LogTest.class, "RCS_Logging", "This is an FATAL
message", m_obj,Syslog.FATAL);

}}
```

The following is a StartupRcs sample code.

```
package gov.ed.sfa.ita.common;

import com.protomatter.syslog.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
/**
 * Insert the type's description here.
 * Creation date: (7/31/2001 8:01:48 PM)
 * @author: Administrator
 */
public class StartupRcs implements
com.ibm.ejs.sm.server.ServiceInitializer {
      private static boolean s_configured = false;
/**
 * StartupRcs constructor comment.
 */
public StartupRcs() {
      super();
}
/**
 * initialize method comment.
 */
public void initialize(Context arg1) throws Exception {
startupSyslog();
}
/**
 * Insert the method's description here.
 * Creation date: (7/31/2001 11:06:44 PM)
 */
private void startupSyslog() {
    try {
        if (s_configured) {
              System.out.println("StartupRcs: Found the Syslog variable
already configured ");
            return;
        }
        this.s_configured = true;
        // get the path to the config file from the
        // "Syslog.config.xml" system property.
        String xmlConfigFile = System.getProperty("Syslog.config.xml");
        if (xmlConfigFile == null) {
              System.out.println("StartupRcs: The xmlConfigFile System
Property is null ");
            return;
        }
        System.out.println("StartupRcs: Configuring Syslog from \"" +
xmlConfigFile+"\"");
        Syslog.configure(new File(xmlConfigFile));
        Iterator loggers = Syslog.getLoggers();
        while (loggers.hasNext()) {
              Syslogger logger = (Syslogger)loggers.next();
              System.out.println("StartupRcs: logger \"" +
logger.getName()  + "\" is a " + logger.getClass().getName() );
        } }catch (Exception x) {
        return;
    }

    return;
}
/**
 * terminate method
 */
public void terminate(Context arg1) throws Exception {
          Syslog.shutdown();
          }
```

5.3.7 Resources

The following resources have more information about the logging service.

- Protomatter site: http://www.protomatter.com
- ITA Logging Design Document

6 RCS – Persistence Framework

The ITA persistence framework provides a transparent and flexible mapping of the business objects to relational database tables. It is transparent in that once the business objects and their mappings are defined, application developers do not need to have any knowledge of the underlying relational database tables. It is flexible in that if the underlying relational database model changes, the business object model does not have to change with it – a change in the mapping layer is all that should be required. The framework is made up of several components working together:

- Domain Component
- Unit of Work Component
- Result Set Component
- Business Mapper Component
- Business Object Component
- Persistable Object Manager (POM) Component

The goal of this development was to provide a simple yet robust persistence framework that could easily be utilized by any SFA development team building applications in Java, or more specifically on the WebSphere Application Server. It should be noted, however, that while the design of the persistence framework does incorporate the best practices as recommended in WebSphere documentation, there is nothing in this package that ties it to WebSphere. An important part of any project, database access is one of the most time-consuming coding tasks and one of the most resource intensive components of a deployed application. Correct database access coding is critical to the performance and maintainability of an application. However, without a framework in place, each developer on a project will code database access as he or she sees fit (and best knows how), often leading to a haphazard implementation and duplication of effort. As such, object persistence and database access should be addressed with care and forethought.

6.1 Testing Conditions & Results

6.1.1 Automated Testing

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	constructor with no arguments	SFADomainTests	testSFADomain	SFADomain	SFADomain()	SFADomain object is created	
2	constructor with a string argument	SFADomainTests	testSFADomain	SFADomain	SFADomain (String pDataSourceName)	SFADomain object is created, setting the dataSourceName field to "testDataSource"	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
3	constructor with three string arguments	SFADomainTests	testSFADomain	SFADomain	SFADomain (String pDataSourceName, String pUserName, String pPassword)	SFADomain object is created, setting the dataSourceName field to "testDataSource", userName to "testUserName", and password to "testPassword"	
4	set and get of dataSourceName field	SFADomainTests	testDataSourceName	SFADomain	setDataSourceName getDataSourceName	the string "testDataSource" is set and retrieved	
5	set and get of userName field	SFADomainTests	testUserName	SFADomain	setUserName getUserName	the string "testUserName" is set and retrieved	
6	set and get of password field	SFADomainTests	testPassword	SFADomain	setPassword getPassword	the string "testPassword" is set and retrieved	
7	constructor called with correct data source name, user name, and password	SFAUnitOfWorkTests	testSFAUnitOfWork	SFAUnitOfWork	SFAUnitOfWork(String pDataSourceName, String pUserName, String pPassword)	SFAUnitOfWork is created with connection to the database	
8	constructor called with invalid data source name	SFAUnitOfWorkTests	testSFAUnitOfWork	SFAUnitOfWork	SFAUnitOfWork(String pDataSourceName, String pUserName, String pPassword)	SFAUnitOfWork is created, but without a connection to the database. An SFAException is thrown.	
9	constructor called with invalid username	SFAUnitOfWorkTests	testSFAUnitOfWork	SFAUnitOfWork	SFAUnitOfWork(String pDataSourceName, String pUserName, String pPassword)	SFAUnitOfWork is created, but without a connection to the database. An SFAException is thrown.	
10	constructor called with correct data source name and user name, but invalid password	SFAUnitOfWorkTests	testSFAUnitOfWork	SFAUnitOfWork	SFAUnitOfWork(String pDataSourceName, String pUserName, String pPassword)	SFAUnitOfWork is created, but without a connection to the database. An SFAException is thrown.	
11	reference to JDBC connection obtained when UnitOfWork properly created	SFAUnitOfWorkTests	testGetConnection	SFAUnitOfWork	getConnection()	valid JDBC connection obtained	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
12	reference to JDBC connection not obtained when UnitOfWork is invalid	SFAUnitOfWorkTests	testGetConnection	SFAUnitOfWork	getConnection()	null returned	
13	executeQuery successful for correctly constructed SQL query	SFAUnitOfWorkTests	testExecuteQuery	SFAUnitOfWork	executeQuery()	SFAResultSet returned with appropriate data	
14	executeQuery unsuccessful for correctly constructed SQL query	SFAUnitOfWorkTests	testExecuteQuery	SFAUnitOfWork	executeQuery()	SFAException raised, no data returned	
15	executeUpdate successful for correctly constructed SQL update statement	SFAUnitOfWorkTests	testExecuteUpdate	SFAUnitOfWork	executeUpdate()	number of rows updated is returned; matches the criteria specified for the query	
16	executeUpdate updates no rows for correctly constructed SQL update statement	SFAUnitOfWorkTests	testExecuteUpdate	SFAUnitOfWork	executeUpdate()	number of rows updated is 0; 0 is the returned value	
17	executeUpdate unsuccessful for incorrectly constructed SQL query	SFAUnitOfWorkTests	testExecuteUpdate	SFAUnitOfWork	executeUpdate()	no rows updated; SFAException is thrown	
18	executeUpdate successful for correctly constructed SQL update statement and abort() called	SFAUnitOfWorkTests	testAbort	SFAUnitOfWork	executeUpdate(), abort()	rows updated matches the criteria specified for the query, but when abort is called the updates are rolled back	
19	executeUpdate successful for correctly constructed SQL update statement and commit() called	SFAUnitOfWorkTests	testCommit	SFAUnitOfWork	executeUpdate(), commit()	rows updated matches the criteria specified for the query, and when commit is called the updates are set in the database	
20	executeUpdate successful for correctly constructed SQL update statement and end() called	SFAUnitOfWorkTests	testEnd	SFAUnitOfWork	executeUpdate(), end()	rows updated matches the criteria specified for the query, and when end is called the updates are set in the database	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
21	SFAResultSet constructed successfully	SFAResultSetTest	testSFAResultSet	SFAResultSet	SFAResultSet(RecordSet rs)	SFAResultSet constructed from executeQuery method of SFAUnitofWork	
22	getBoolean successful for valid input column name	SFAResultSetTest	testGetBoolean	SFAResultSet	getBoolean(String)	correct value returned for row (row checked by verifying against row id)	
23	getBoolean fails for invalid input column name	SFAResultSetTest	testGetBoolean	SFAResultSet	getBoolean(String)	value not returned; SFAException raised	
24	getDate successful for valid input column name	SFAResultSetTest	testGetDate	SFAResultSet	getDate(String)	correct value returned for row (row checked by verifying against row id)	
25	getDate fails for invalid input column name	SFAResultSetTest	testGetDate	SFAResultSet	getDate(String)	value not returned; SFAException raised	
26	getFloat successful for valid input column name	SFAResultSetTest	testGetFloat	SFAResultSet	getFloat(String)	correct value returned for row (row checked by verifying against row id)	
27	getFloat fails for invalid input column name	SFAResultSetTest	testGetFloat	SFAResultSet	getFloat(String)	value not returned; SFAException raised	
28	getInt successful for valid input column name	SFAResultSetTest	testGetInt	SFAResultSet	getInt(String)	correct value returned for row (row checked by verifying against row id)	
29	getInt fails for invalid input column name	SFAResultSetTest	testGetInt	SFAResultSet	getInt(String)	value not returned; SFAException raised	
30	getLong successful for valid input column name	SFAResultSetTest	testGetLong	SFAResultSet	getLong(String)	correct value returned for row (row checked by verifying against row id)	
31	getLong fails for invalid input column name	SFAResultSetTest	testGetLong	SFAResultSet	getLong(String)	value not returned; SFAException raised	
32	getObject successful for valid input column name	SFAResultSetTest	testGetObject	SFAResultSet	getObject(String)	correct value returned for row (row checked by verifying against row id)	
33	getObject fails for invalid input column name	SFAResultSetTest	testGetObject	SFAResultSet	getObject(String)	value not returned; SFAException raised	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
34	getString successful for valid input column name	SFAResultSetTest	testGetString	SFAResultSet	getString(String)	correct value returned for row (row checked by verifying against row id)	
35	getString fails for invalid input column name	SFAResultSetTest	testGetString	SFAResultSet	getString(String)	value not returned; SFAException raised	
36	next moves recordset to next row	SFAResultSetTest	testNext	SFAResultSet	next	true returned; recordset moved to next row; validated by checking the next row id against the database	
37	next called at the end of the recordset	SFAResultSetTest	testNext	SFAResultSet	next	false returned; recordset not moved	
38	close called and succesful	SFAResultSetTest	testClose	SFAResultSet	close	nothing returned; no exceptions raised	
39	close called and fails	SFAResultSetTest	testClose	SFAResultSet	close	exception raised because recordset is already closed	
40	constructor for POM successful with Domain	SFAPersistableObjectManagerTest	testSFAPersistableObjectManager	SFAPersistableObjectManager	SFAPersistableObjectManager(SFADomain domain)	SFAPersistableObjectManager created	
41	constructor for POM successful with UnitOfWork	SFAPersistableObjectManagerTest	testSFAPersistableObjectManager	SFAPersistableObjectManager	SFAPersistableObjectManager(SFAUnitOfWork uow)	SFAPersistableObjectManager created	
42	addObject successful for query string and parameters	SFAPersistableObjectManagerTest	testAddObject	SFAPersistableObjectManager	addObject(String sql, Vector vals)	row(s) added to database	
43	getObject successful for query string and parameters	SFAPersistableObjectManagerTest	testGetObject	SFAPersistableObjectManager	getObject(ISFAPersistableMapper ipm, String query, Vector vals)	data retrieved from database	
44	getObjects successful for query string and parameters	SFAPersistableObjectManagerTest	testGetObjects	SFAPersistableObjectManager	getObjects(ISFAPersistableMapper ipm, String query, Vector vals)	rows retrieved from database, returned as SFAResultSet	
45	getObjectsAsHashtable successful for query string and parameters	SFAPersistableObjectManagerTest	testGetObjectsAsHashtable	SFAPersistableObjectManager	getObjectsAsHashtable(ISFAPersistableMapper ipm, String query, Vector vals)	rows retrieved from database, returned as hashtable	
46	removeObject successful for query string and parameters	SFAPersistableObjectManagerTest	testRemoveObject	SFAPersistableObjectManager	removeObject(String query, Vector vals)	row(s) deleted from database	
47	removeObjects successful for query string and parameters	SFAPersistableObjectManagerTest	testRemoveObjects	SFAPersistableObjectManager	removeObjects(String query, Vector vals)	rows deleted from database	
48	updateObject successful for query string and parameters	SFAPersistableObjectManagerTest	testUpdateObject	SFAPersistableObjectManager	updateObject(String query, Vector vals)	row(s) updated in database	

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
49	updateObjects successful for query string and parameters	SFAPersistableObjectManagerTest	testUpdateObjects	SFAPersistableObjectManager	updateObjects(String query, Vector vals)	row(s) updated in database	
50	abortTransaction successful	SFAPersistableObjectManagerTest	testAbortTransaction	SFAPersistableObjectManager	abortTransaction()	updated started, but no rows changed	
51	commitTransaction successful	SFAPersistableObjectManagerTest	testCommitTransaction	SFAPersistableObjectManager	commitTransaction()	update started and completed	
52	end Transaction successful	SFAPersistableObjectManagerTest	testEndTransaction	SFAPersistableObjectManager	endTransaction()	update started and completed	

6.2 Performance Analysis

6.2.1 Summary

A careful review of the Persistence framework revealed no performance deficiencies in its design or code. The framework is very lightweight, imposing minimal overhead on what could be accomplished with a straight JDBC implementation of database access. In fact, in many cases the code would most likely outperform straight JDBC access because the framework supports best practices for database access that might be overlooked by Java developers who are inexperienced with JDBC. Performance metrics further underline the capabilities of the framework.

Some areas of consideration were noted, however. These areas did not involve deficiencies in the framework itself. Instead it involved framework configurations and usages that would lead to slow performance. Two areas in particular were business object design and SQL coding. Business objects can contain validation code; therefore poorly designed code can lead to slow object retrievals and updates. While it is good to have objects validated before they are persisted, care should be taken to make the objects as lightweight as possible. Mapping objects contain all of the SQL code used to persist objects to the database; therefore poorly designed SQL code and database design would have an adverse impact on framework performance. For example, doing object retrievals on a non-indexed field would require a full table scan for each object retrieved.

These issues are not unique to the Persistence framework, however. Poor validation and SQL code are common areas of concern when writing database access code. The ITA Persistence framework actually serves to minimize the occurrence of these issues. The design of the framework results in that:

- JDBC code is written once, delivered in the framework itself
- Validation code is written once, by the lead developer who is most skilled in that area
- SQL code is written once, by the lead developer who is most skilled in that area

Therefore, a properly implemented Persistence framework should not only speed development, but improve the quality and performance of the code.

6.2.2 Background

The framework is fairly lightweight; it incurs a minimal performance penalty over a straight JDBC implementation. As described in the design document and user's guide, however, the benefits from a object based design for data access more than make up for the small performance hit. The impact of the various objects on the overall performance is outlined below:

The Domain component has no effect; it simply stores connection information to the database and is used only once, when the connection is established.

The Unit of Work component contains all of the JDBC code for the framework. It is very straightforward code, uses standard and accepted JDBC practices for database access.

The Result Set component is a light wrapper of the JDBC ResultSet. Its main function is to format values pulled from the ResultSet in appropriate Java datatypes.

The Business Mapper component contains all of the SQL code for database interaction; poorly designed SQL code could lead to degradation in performance.

The Business Object component is the Java representation of the data in the database tables. It may also contain validation code; a poorly designed business object could lead to degradation in performance.

The POM component uses all of the above objects to coordinate persistence activities. The code there is provided with the framework and does not need to be modified by application developers. It is also fairly lightweight and straightforward and should incur no appreciable performance hit.

6.2.3 Testing Environment

The testing harness was run on a standard SFA developer workstation. The hardware consisted of a Compaq Deskpro with a single 600 MHz Pentium III processor and 512 MB of RAM. The machine ran Windows NT 4.0 Service Pack 6. The execution environment was IBM's WebSphere application server, version 3.5.3, running on IBM's JDK 1.2.2. This same machine was also running a full instance of an Oracle database, version 8.0.6.0.0. While tests were run, no other applications were loaded into memory, and the system was not interacted with. This was done in order to leave all resources available to the test harness, and eliminate the possibility of unexplained behavior in the tables of results.

6.2.4 Testing Configuration

There is very little configuration that needs to be done for the ITA Persistence framework itself. The only system level property that needs to be configured is the JDBC DataSource name. This is done in the WebSphere administration console.

JProbe needs to be configured to be able to gather metrics from the framework as it ran in the IBM Websphere environment. There are three components to this configuration. One is the profleWAS30.jpl file, which lists all profiling parameters in an XML file. These parameters are normally set in the JProbe GUI, but since a server is being monitored, they are set through a file interface. The text of the configuration file is provided below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
      <program type="application">
            <application
                  args=""
                  working_dir=""
                  source_dir=""
                  classname="">
                  <classpath>
                        <classpath.path
location="%CLASSPATH%"/>
                  </classpath>
            </application>
            <applet
                  working_dir=""
                  source_dir=""
                  htmlfile=""
                  main_package="">
                  <classpath>
                        <classpath.path
location="%CLASSPATH%"/>
                  </classpath>
            </applet>
            <serverside
                  suggested_filters=""
                  id=""
                  server_dir=""
                  prepend_to_vm_args=""
                  source_dir=""
                  classname=""
                  main_package=""
                  exclude_server_classes="true"
                  args=""
                  working_dir=""
                  prepend_to_classpath="">
                  <classpath>
                        <classpath.path
location="%CLASSPATH%"/>
                  </classpath>
            </serverside>
      </program>
      <vm
            snapshot_dir="C:\Temp"
            location="E:\WebSphere\AppServer\jdk\bin\java.exe"
            args=""
            type="java2"
            use_jit="true"/>
      <viewer
            socket=":4444"
            type="local"/>
      <analysis type="profile">
            <performance
                  record_from_start="true"
                  timing="elapsed"
                  track_natives="true"
                  final_snapshot="true"
```

```
                        granularity="method">
                        <performance.filter
                                visibility="visible"
                                methodmask="*"
                                enabled="true"
                                classmask="*"
                                time="ignore"
                                granularity="method"/>
                        <performance.filter
                                visibility="visible"
                                methodmask="*"
                                enabled="true"
                                classmask=".*"
                                time="track"
                                granularity="method"/>
                </performance>
                <heap
                        record_from_start="true"
                        no_stack_trace_limit="false"
                        final_snapshot="false"
                        max_stack_trace="4"
                        track_dead_objects="false"/>
                <threadalyzer
                        record_from_start="true"
                        write_to_console="false">
                        <deadlock_detection
                                enabled="true"
                                deadlock_and_exit="true"
                                report_stalls="false"
                                track_system_threads="false"
                                block_can_stall="false"
                                deadlock_threshold="2"/>
                        <deadlock_prediction
                                enable_hold_and_wait="false"
                                enable_lock_order="false"
                                lock_order_maintains_covers="true"/>
                        <data_race
                                ignore_volatile="false"
                                enable_happens_before="false"
                                no_stack_trace_limit="false"
                                enable_lock_covers="false"
                                max_stack_trace="1"
                                instrument_elements="false"/>
                        <visualizer
                                enabled="true"
                                visualization_level="1"/>
                        <threadalyzer.filter
                                visibility="invisible"
                                enabled="true"
                                classmask="*"/>
                        <threadalyzer.filter
                                visibility="visible"
                                enabled="true"
                                classmask=".*"/>
                </threadalyzer>
                <coverage
                        record_from_start="true"
                        final_snapshot="true"
```

```
                  granularity="line">
                  <coverage.filter
                        visibility="invisible"
                        methodmask="*"
                        enabled="true"
                        classmask="*"/>
                  <coverage.filter
                        visibility="visible"
                        methodmask="*"
                        enabled="true"
                        classmask=".*"/>
            </coverage>
      </analysis>
</jpl>
```

The other three configuration issues occur in WebSphere itself. In the administration console, for the server being evaluated, (usually Default Server):

1. The name must be changed to eliminate spaces – either Default_Server or DefaultServer is a good option

2. The environment must include the following variables –
   EXECUTABLE=[directory where JProbe is installed]\jplauncher.exe EXECUTE=YES

3. The command line arguments for the server should be updated to add
   -jp_input= <path_of_JProbe_logfile> / profileWAS30.jpl

Consult the JProbe website (www.jprobe.com / j2ee) for more detailed instructions on integrating the testing environment with the IBM WebSphere application server.

6.2.5 Testing Scenarios

The persistence performance test focused on two usage scenarios for analysis. The first was persistence of objects to the database, and the second was retrieval of objects from the database. These usage scenarios cover all major components of the framework, and should provide a good indication of the general performance characteristics of the system.

The updates and retrievals were both executed in runs of 1, 10, and 1000 iterations. These were conducted to see if performance was linear or degraded as the load was increased. This also improved the chances of spotting over-allocation or lack of freeing up objects.

The model used was the OrderRequest Business Object and Mapper. It is a very straightforward implementation but is a very real-world example of what the framework could be used to do. The implementation of these objects was described in detail in the ITA Persistence User Guide (Section 6.3).

6.2.6 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: To identify any performance bottleneck due to inefficient code algorithms

6.2.6.1 Memory (Heap) Usage

The performance test utilized JProbe Profiler's Memory Debugger to identify the parts of the framework that might be causing loitering objects. This was accomplished by analysis of the Java heap. The Runtime Heap Summary window can be used to view instance counts and information on allocating methods.

The Heap Usage Chart below plots the size of the Java heap over the course of two distinct test runs. The runs consist of an object update run and an object retrieval run. Each graph captures heap usage from the startup and initialization of the WebSphere server through the end of the run.. The chart helps to visualize memory use in the Java heap. It displays the available size of the Java heap (in pink) and the used memory (in blue) over time.

The first part of each chart indicates a rising graph. A rising graph in these charts indicates that the core set of WebSphere application server objects are being initialized for use. This is the expected behavior of the application server.

Test runs were not initiated until the application server was fully initialized and had reached a steady state, indicated by a steady graph in the heap usage chart. These steady graphs are visible roughly in the middle of the chart.

The third part of the chart indicates a series of spikes. Steep spikes in the Heap Usage Chart represent temporary objects being allocated and garbage collected. If the level of the troughs becomes higher over time, then not all the temporary objects are garbage collected. As can be seen, the troughs for object updates remain steady over time, and while multiple objects are being created and destroyed, there do not appear to be any lingering objects. The troughs for object retrievals appear to rise slightly at the end of the run, but that is just because the garbage collection did not occur at the end of the run. As will be visible from the object usage charts on the following pages, all objects instantiated by the Persistence framework were released and garbage collected.

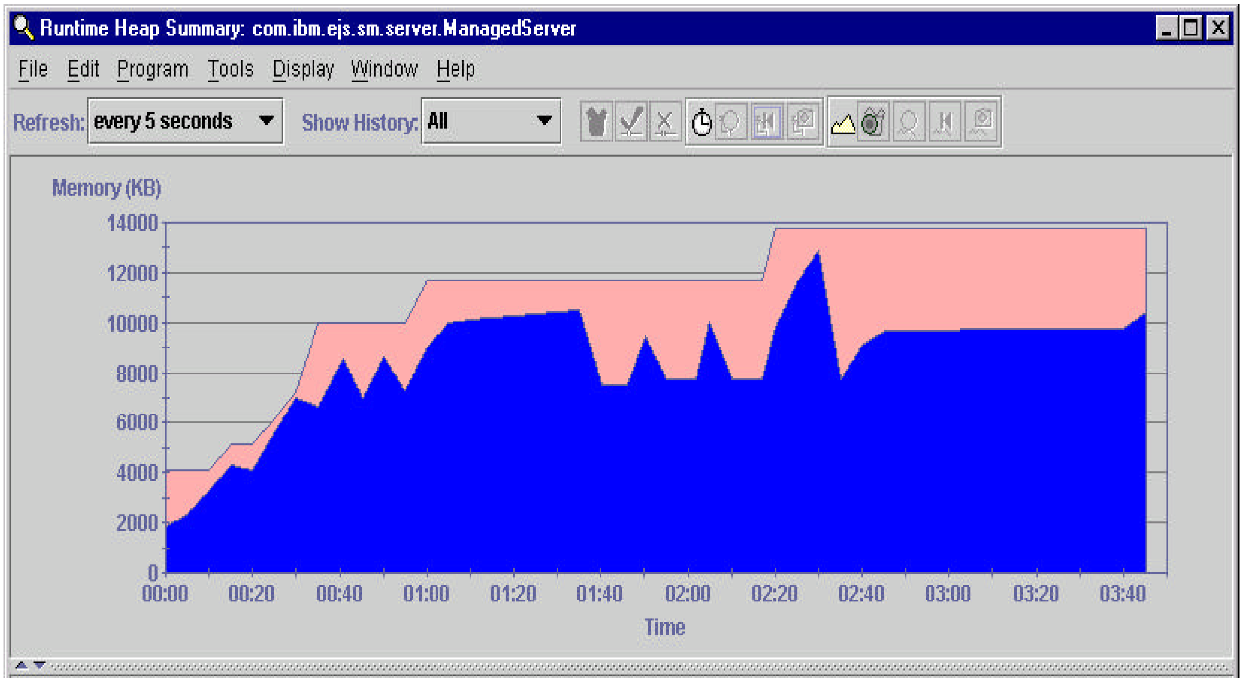


Figure 2. Heap usage covering one object update test run.

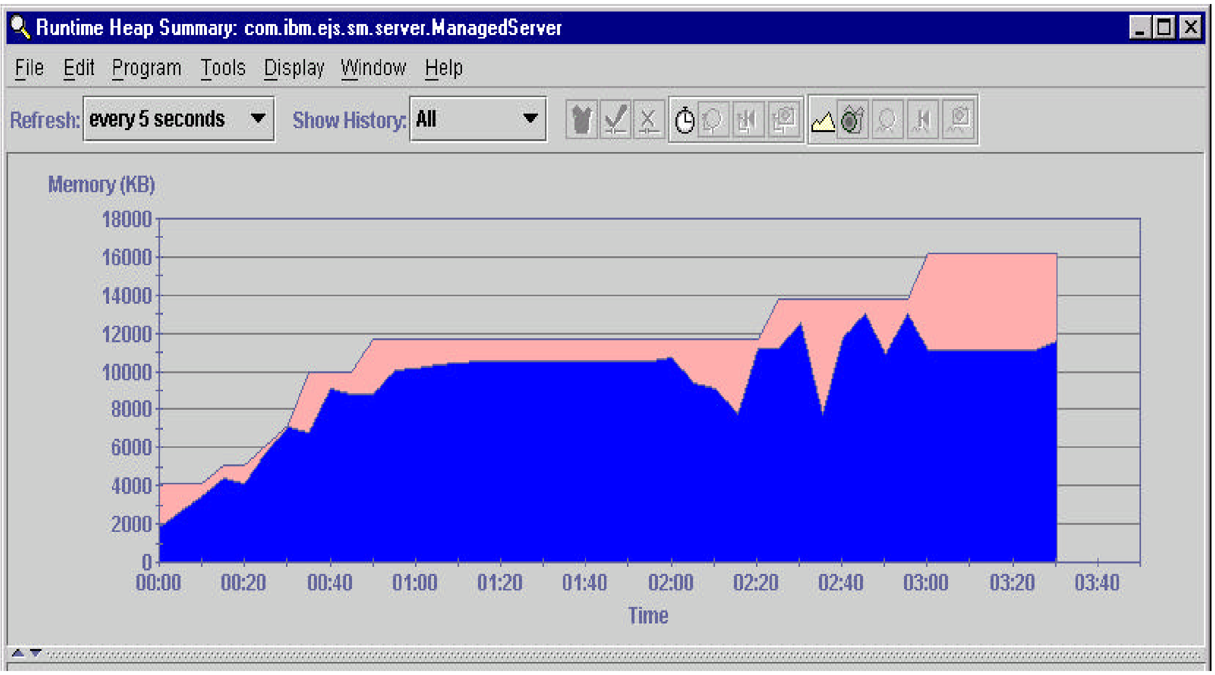


Figure 3. Heap usage covering one object retrieval test run.

6.2.6.2 Garbage Collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage collected objects accumulates. The list below is the final top ten, displaying cumulative objects created at the end of program execution.

Each row identifies the class by package name, if any, and class name. The next columns state, in order, the number of garbage collected objects (GC'ed column) for the class, the number of instances remaining in the heap (Alive column), and the method that allocated the instances of the class (AllocatedAt column). The same class can appear more than once because more than one method allocated instances of the class.

The charts below do not show any unexpected activity, or activity that would indicate a performance problem. Most of the objects created are either strings, string buffers, or character arrays. These numbers are in line with the framework requirements and expected behavior as it formats a large number of messages to write them to the file.

Action: Update Runs: 1

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	9	0	SFAParser.getStatement
java.lang	String	4	2	SFAParameter.setParamType
	char[]	4	2	SFAParameter.setParamType
java.lang	StringBuffer	4	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita.persistence	SFAOracleParser	1	0	SFAPersistableObjectManager.getParser
gov.ed.sfa.ita.persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

Action: Update Runs: 10

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	90	0	SFAParser.getStatement
java.lang	String	40	2	SFAParameter.setParamType
	char[]	40	2	SFAParameter.setParamType
java.lang	StringBuffer	40	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita.persistence	SFAOracleParser	10	0	SFAPersistableObjectManager.getParser

Package	Class	GC'ed	Alive	Allocated At
gov.ed.sfa.ita.persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

Action: update runs: 1000

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	9000	0	SFAParser.getStatement
java.lang	String	4000	2	SFAParameter.setParamType
	char[]	4000	2	SFAParameter.setParamType
java.lang	StringBuffer	4000	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita.persistence	SFAOracleParser	1000	0	SFAPersistableObjectManager.getParser
gov.ed.sfa.ita.persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

Action: Retrieve Runs: 1

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	3	0	SFAParser.getStatement
java.lang	String	1	0	SFAParameter.setParamType
	char[]	1	0	SFAParameter.setParamType
gov.ed.sfa.ita.persistence	SFAOracleParser	1	0	SFAPersistableObjectManager.getParser
java.lang	StringBuffer	1	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita.persistence	SFAResultSet	1	0	SFAUnitOfWork.executeQuery
gov.ed.sfa.ita.persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

Action: Retrieve Runs: 10

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	30	0	SFAParser.getStatement
java.lang	String	10	0	SFAParameter.setParamType
	char[]	10	0	SFAParameter.setParamType
gov.ed.sfa.ita.persistence	SFAOracleParser	10	0	SFAPersistableObjectManager.getParser
java.lang	StringBuffer	10	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita.persistence	SFAResultSet	10	0	SFAUnitOfWork.executeQuery
gov.ed.sfa.ita.persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

Action: Retrieve **Runs: 1000**

Package	Class	GC'ed	Alive	Allocated At
java.lang	StringBuffer	3000	0	SFAParser.getStatement
java.lang	String	1000	0	SFAParameter.setParamType
	char[]	1000	0	SFAParameter.setParamType
gov.ed.sfa.ita. persistence	SFAOracleParser	1000	0	SFAPersistableObjectManager.getParser
java.lang	StringBuffer	1000	0	SFAParser.getStringWrappedValue
gov.ed.sfa.ita. persistence	SFAResultSet	1000	0	SFAUnitOfWork.executeQuery
gov.ed.sfa.ita. persistence	SFAUnitOfWork	1	0	SFAPersistableObjectManager.<init>

6.2.6.3 Code Efficiency

There are nine efficiency metrics that can be collected in Jprobe — five basic metrics and four compound metrics. The basic metrics include Number of Calls, Method Time, Cumulative Time, Method Object Count, Cumulative Object Count. The compound metrics are averages per number of calls, including Average Method Time, Average Cumulative Time, Average Method Object Count, and Average Cumulative Object Count. Time is measured as elapsed time.

The following list defines the nine performance metrics:

- Number of Calls - The number of times the method was invoked.
- Method Time - The amount of time spent executing the method, excluding time spent in its descendants.
- Cumulative Time - The total amount of time spent executing the method, including time spent in its descendants but excluding time spent in recursive calls to descendants.
- Method Object Count - The number of objects created during the method's execution, excluding those created by its descendants.
- Cumulative Object Count - The total number of objects created during the method's execution, including those created by its descendants.
- Average Method Time - Method Time divided by Number of Calls.
- Average Cumulative Time - Cumulative Time divided by Number of Calls.
- Average Method Object - Count Method Object Count divided by Number of Calls.

The charts on the following pages serve to document the performance characteristics of the Persistence framework with the list of methods with the most calls. The data was gathered on runs of 1,000 updates and 1,000 retrievals. This can serve as a basic indicator of processing resource utilization. The list can be reviewed for unexpected activity or optimization opportunities.

Unfortunately, it appears that for test run through an application server, data on time spent in methods, objects allocated in methods, etc., is unavailable. While this data could be useful, it does not invalidate or hamper this analysis. The data could be used to determine bottlenecks in the framework. If the framework were performing poorly than this information would provide the first areas where investigation would provide most benefit. However, the raw performance data in the final section shows that framework performance is quite adequate, so therefore the data does not need to be investigated.

As a result, the following charts serve to document the methods called in the persistence framework. It does validate that the number of calls matches what would be expected for the number of runs conducted (1000).

Object Update Methods from the Persistence framework, ordered by number of calls:

Name	Package	Calls	Source
SFAParameter.getValue()	gov.ed.sfa.ita.persistence	8,000	SFAParameter.java
SFAParser.getParamDelimiter()	gov.ed.sfa.ita.persistence	8,000	SFAParser.java
SFAParameter.<init>(String, String, char)	gov.ed.sfa.ita.persistence	4,000	SFAParameter.java
SFAParameter.getName()	gov.ed.sfa.ita.persistence	4,000	SFAParameter.java
SFAParameter.getParamType()	gov.ed.sfa.ita.persistence	4,000	SFAParameter.java
SFAParameter.setParamType(char)	gov.ed.sfa.ita.persistence	4,000	SFAParameter.java
SFAParameter.setValue(String)	gov.ed.sfa.ita.persistence	4,000	SFAParameter.java
SFAParser.getStringWrappedValue(SFAParameter)	gov.ed.sfa.ita.persistence	4,000	SFAParser.java
SFAParser.getWrappedValue(SFAParameter)	gov.ed.sfa.ita.persistence	4,000	SFAParser.java
SFAOracleParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAOracleParser.java
SFAParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAParser.getStatement()	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAPersistableObjectManager.addObject(String, Vector)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAPersistableObjectManager.commitTransaction()	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java

Name	Package	Calls	Source
SFAPersistableObjectManager.getParser(String)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAPersistableObjectManager.getQuery(String, Vector)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAQueryParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAQueryParser.processRawStatement()	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAQueryParser.setParams(Vector)	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAUnitOfWork.commit()	gov.ed.sfa.ita.persistence	1,000	SFAUnitOfWork.java
SFAUnitOfWork.executeUpdate(String)	gov.ed.sfa.ita.persistence	1,000	SFAUnitOfWork.java
SFADomain.<init>(String, String, String)	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getDataSourceName()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getPassword()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getUserName()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFAPersistableObjectManager.<init>(SFADomain)	gov.ed.sfa.ita.persistence	1	SFAPersistableObjectManager.java
SFAUnitOfWork.<init>(String, String, String)	gov.ed.sfa.ita.persistence	1	SFAUnitOfWork.java

Object Retrieval Methods from the Persistence framework, ordered by number of calls:

Name	Package	Calls	Source
SFAResultSet.getString(String)	gov.ed.sfa.ita.persistence	4,000	SFAResultSet.java
SFAParameter.getValue()	gov.ed.sfa.ita.persistence	2,000	SFAParameter.java
SFAParser.getParamDelimiter()	gov.ed.sfa.ita.persistence	2,000	SFAParser.java
SFAOracleParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAOracleParser.java
SFAParameter.<init>(String, String, char)	gov.ed.sfa.ita.persistence	1,000	SFAParameter.java
SFAParameter.getName()	gov.ed.sfa.ita.persistence	1,000	SFAParameter.java
SFAParameter.getParamType()	gov.ed.sfa.ita.persistence	1,000	SFAParameter.java
SFAParameter.setParamType(char)	gov.ed.sfa.ita.persistence	1,000	SFAParameter.java
SFAParameter.setValue(String)	gov.ed.sfa.ita.persistence	1,000	SFAParameter.java
SFAParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAParser.getStatement()	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAParser.getStringWrappedValue(SFAParameter)	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAParser.getWrappedValue(SFAParameter)	gov.ed.sfa.ita.persistence	1,000	SFAParser.java
SFAPersistableObjectManager.getObject(ISFAPersistableMapper, String, Vector)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java

Name	Package	Calls	Source
SFAPersistableObjectManager.getParser(String)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAPersistableObjectManager.getQuery(String, Vector)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAPersistableObjectManager.nextObject(SFAResultSet, ISFAPersistableMapper)	gov.ed.sfa.ita.persistence	1,000	SFAPersistableObjectManager.java
SFAQueryParser.<init>(String)	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAQueryParser.processRawStatement()	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAQueryParser.setParams(Vector)	gov.ed.sfa.ita.persistence	1,000	SFAQueryParser.java
SFAResultSet.<init>(ResultSet, Statement)	gov.ed.sfa.ita.persistence	1,000	SFAResultSet.java
SFAResultSet.close()	gov.ed.sfa.ita.persistence	1,000	SFAResultSet.java
SFAResultSet.next()	gov.ed.sfa.ita.persistence	1,000	SFAResultSet.java
SFAUnitOfWork.executeQuery(String)	gov.ed.sfa.ita.persistence	1,000	SFAUnitOfWork.java
SFADomain.<init>(String, String, String)	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getDataSourceName()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getPassword()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFADomain.getUserName()	gov.ed.sfa.ita.persistence	1	SFADomain.java
SFAPersistableObjectManager.<init>(SFADomain)	gov.ed.sfa.ita.persistence	1	SFAPersistableObjectManager.java

Name	Package	Calls	Source
SFAUnitOfWork.<init>(String, String, String)	gov.ed.sfa.ita.persistence	1	SFAUnitOfWork.java

6.2.7 General Performance Metrics

The ITA Persistence framework was run through a performance testing harness to determine system-level performance metrics for the framework components. The harness tested the system as a whole, for object updates and object retrievals, in the WebSphere application server environment. Running the tests on a Compaq Deskpro with an Intel 600 MHz Pentium III processor, running WindowsNT 4.0 and using WebSphere 3.5.3 and Oracle 8.0.6.0.0 yields the following performance numbers:

ITA Persistence performance statistics from Intel desktop test machine

Operation	Iterations	Total Time	Average Time	Calls Per Second
Object Update	1000	5,900 ms	5.9 ms	169.5
Object Retrieval	1000	4,116 ms	4.1 ms	243.9

As the chart above indicates, the ITA Persistence framework has strong performance characteristics and is capable of performing 100-200 persistence operations a second on the test machine. It should be noted that these servers, were not tuned for performance in any way. In particular, the full version of Oracle running on the desktop test machine probably most affected the performance. On production level machines, with tuned servers, these numbers would increase significantly, and it should meet the needs of most applications. It incurs minimal processing over what would be required for a normal JDBC implementation and provides a solid, stable framework on which to conduct database transactions.

6.3 User Guide

6.3.1 Introduction

6.3.1.1 Purpose

This section provides a high-level summary and usage scenarios for the Integrated Technical Architecture (ITA) standard Persistence Framework. The Persistence framework is part of a suite of frameworks called Reusable Common Services (RCS), which are provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects.

6.3.1.2 Intended audience

This section is intended for ITA and SFA application programmers who need to understand the Persistence framework in order to use this framework in their applications.

Primarily, one senior developer on a team will need to know the Persistence framework in detail in order to create mappers and business objects, while the other application programmers will only need to know how to use the mappers and business objects in their application code. More detail on user responsibilities will be presented in the usage scenarios.

6.3.2 Background

The goal of the development of this ITA Persistence framework was to provide a simple yet robust toolset that could easily be utilized by any SFA development team building applications in Java, or more specifically on the WebSphere Application Server. It should be noted, however, that while the design of the persistence framework does incorporate the best practices as recommended in WebSphere documentation, there is nothing in this package that ties it to WebSphere.

Database access, an important part of any project, is one of the most time-consuming coding tasks and one of the most resource intensive components of a deployed application. Correct database access coding is critical to the performance and maintainability of an application. However, without a framework in place, each developer on a project will code database access as he or she sees fit (and best knows how), often leading to a haphazard implementation and duplication of effort. Some planning and forethought up front can save an application development team countless hours of tracing and debugging incorrect or inefficient database access.

6.3.2.1 Scope

This section covers only installation, features, and usage scenarios of the ITA Persistence framework. The Persistence framework is tied to the ITA Exception Handling framework, and typically used with the ITA Logging framework – however, the Exception Handling and Logging frameworks are not covered in this section. Consult the ITA Exception Handling and ITA Logging user guides for more information on these frameworks.

6.3.2.2 Assumptions

The Persistence framework has been designed and tested in a J2EE application server environment. Specifically, it was developed in the current production environment for SFA: IBM WebSphere 3.5, running on its required IBM JDK version 1.2.2. It uses Java Database Connectivity (JDBC) 2.0, with the database pooling implementation of the server. It should also work with the current JavaServer Pages (1.1), Java Servlet (2.2), and Java Messaging Service (1.0.1) specifications for this server. It was built and tested on Sun Solaris 2.6 operating system.

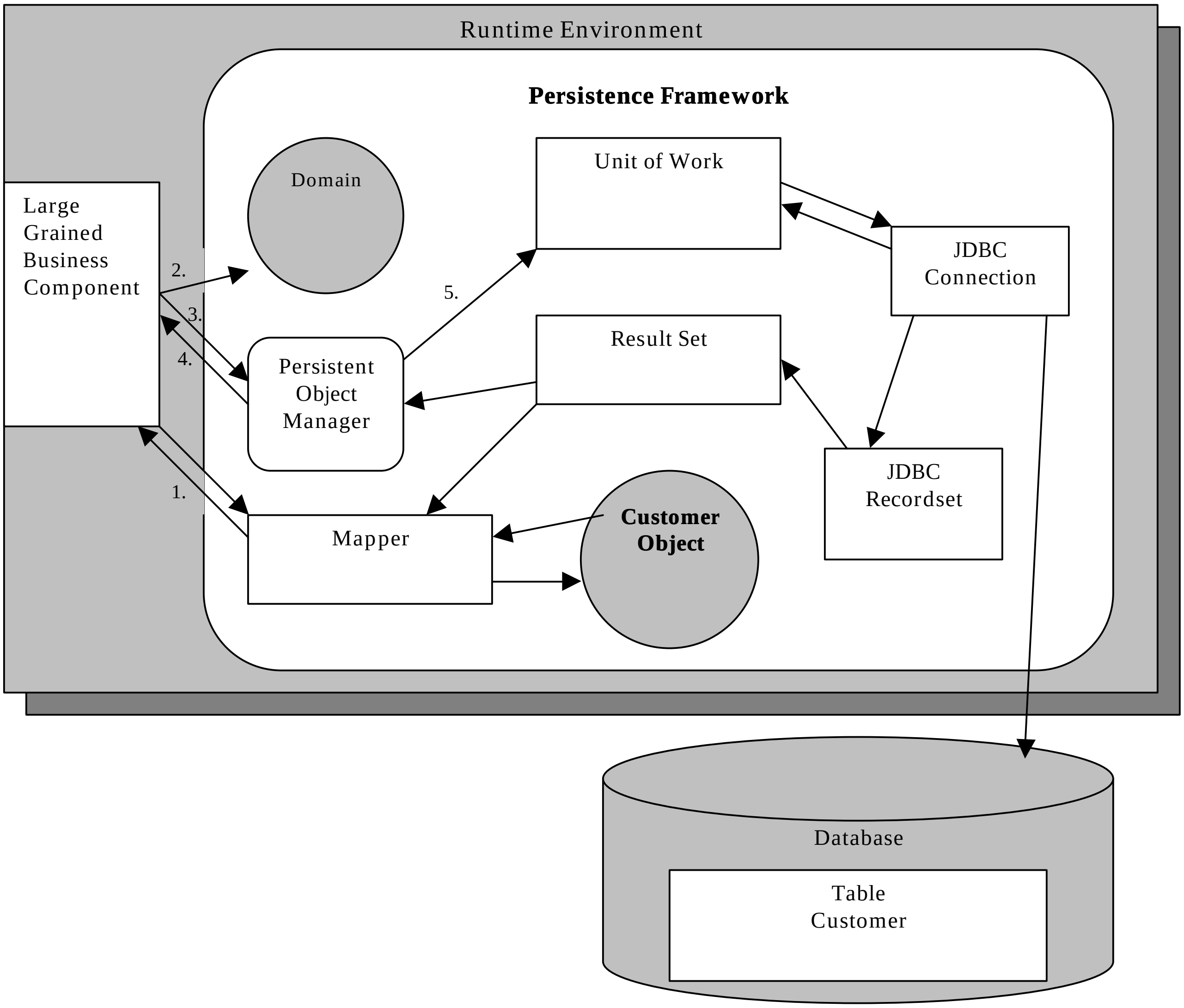
6.3.3 Description

6.3.3.1 Overview

The ITA persistence framework provides a transparent and flexible mapping of the business objects to relational database tables. It is transparent in that once the business objects and their mappings are defined; application developers do not need to have any knowledge of the underlying relational database tables. It is flexible in that if the underlying relational database model changes, the business object model does not have to change with it – a change in the mapping layer is all that should be required. The framework is made up of several components working together:

- Domain Component
- Unit of Work Component
- Persistable Object Manager Component
- Result Set Component
- Business Mapper Component

- Business Object Component



A diagram of the components working together (retrieving a Customer Object from the database) is presented below:

Explanation of Steps:

1. Create Mapper and Business Object, populate values to update
2. Create Domain with connection information
3. Create Persistable Object Manager passing the domain
4. Request object from the POM passing the Mapper information
5. POM executes the query, maps the data to the object, closes the unit of work, and returns the object

6.3.3.2 Features

The ITA persistence feature set is designed to hide database access complexities from application developers. By abstracting the business object representation of data from its physical database storage characteristics, this framework provides features such as:

- Eliminating the need for application developers to be knowledgeable about the system database design
- Eliminating the need for application developers to be proficient in SQL
- Consolidating all SQL calls in one place to maximize performance and ease maintenance
- Eliminating the need for application developers to be proficient with the JDBC library
- Consolidating data validation rules in the business object layer to ensure data integrity is consistently maintained

In addition, the framework was designed to meet SFA application development requirements, and to work optimally in the SFA technical environment. Specifically, the framework has been designed to:

- Support applications running on the IBM WebSphere Application Server (WAS)
- Implement IBM WAS best practices for database access
- Implement IBM WAS best practices for object management
- Integrate the ITA exception handling framework
- Support ad-hoc database interaction

6.3.3.3 Main Concepts

The ITA Persistence framework consists of the following main classes:

Domain

The Domain is an object store that holds a database URL, a username, and a password. It represents the domain that the data resides in for the application. Storing database connection information in this manner supports ease of application configuration updates.

Unit of Work

The UnitOfWork models a persistence unit of work. A UnitOfWork has a Connection that represents the actual physical connection to the database. It allows a series of related actions to be logically grouped together that can be either committed or rolled back against a database connection.

Persistable Object Manager

The Persistable Object Manager is used to create, read, update, and delete information from the data store. It holds (or creates and holds) a single Unit of Work that can be used throughout a Business Component method that accesses the data store.

Methods from the BusinessComponent class and the Business class that call methods within this class should create a single PersistableObjectManager object and directly call the methods here for persistent data access and storage. The PersistableObjectManager class also acts as a wrapper for the data access layer (JDBC).

Result Set

The Result Set is a wrapper around the JDBC ResultSet to provide transparent access to data stored in the result set. It allows the architecture to access the result set in an object-oriented manner. For example, sending next() to the result set will return the next business object instance in the result set. Executing database operations such as select and detect on the Extent will return a Result Set of business object instances retrieved from the data store.

Business Mapper

The Business Mapper needs to provide the Business Object – specific mapping information to the rest of the framework. The ISFAPersistableMapper interface defines what methods will be required for a functional persistable business object mapper to implement. Implementing the interface for a particular business object will lead to a Mapper class, where all database specific code (select, insert, update, delete statements, etc.) will be contained. The framework then uses this code to map the business object to the database.

Business Object

The Business Object is a basic data structure with very little code overhead other than private data members and public set() and get() methods to access them. In this way the data that is retrieved from the database is completely decoupled from its Java representation. As a result once data is retrieved it can be manipulated with very little overhead (i.e. database access or connections) until it needs to be persisted back to the data store.

6.3.4 Installation

6.3.4.1 Software requirements

The ITA Persistence framework was designed and tested in SFA's production server environment, IBM WebSphere 3.5.3 and on IBM's JDK 1.2.2. It can only run in the WebSphere environment as it utilizes WebSphere Data Sources for its database connectivity. The Persistence framework requires a database; a standard relational database, such as Oracle, SQL Server, or DB2, must be accessible from the WebSphere server. An appropriate JDBC driver for the system database will have to be installed on the WebSphere server and set in the classpath.

The ITA Persistence framework also makes use of the ITA Exception Handling framework v1.5. Please see the user guide for the Exception Handling framework for instructions on its installation and configuration. If applications intend to log error messages using the ITA Logging framework, then the user guide for that framework should be consulted for the appropriate configuration and supporting libraries.

6.3.4.2 Installation procedures

The Persistence framework uses one jar file, rcs_persistence_v1.5.jar, for operation. This jar file contains all necessary core classes for proper execution of the Persistence framework. The classpath of the server should be updated to refer to this file. As the Persistence framework makes use of the ITA Exception Handling framework, that jar file, rcs_exception_v1.5.jar, should also be installed (copied to the same directory as the persistence jar file) and the classpath should also be updated to refer to that file.

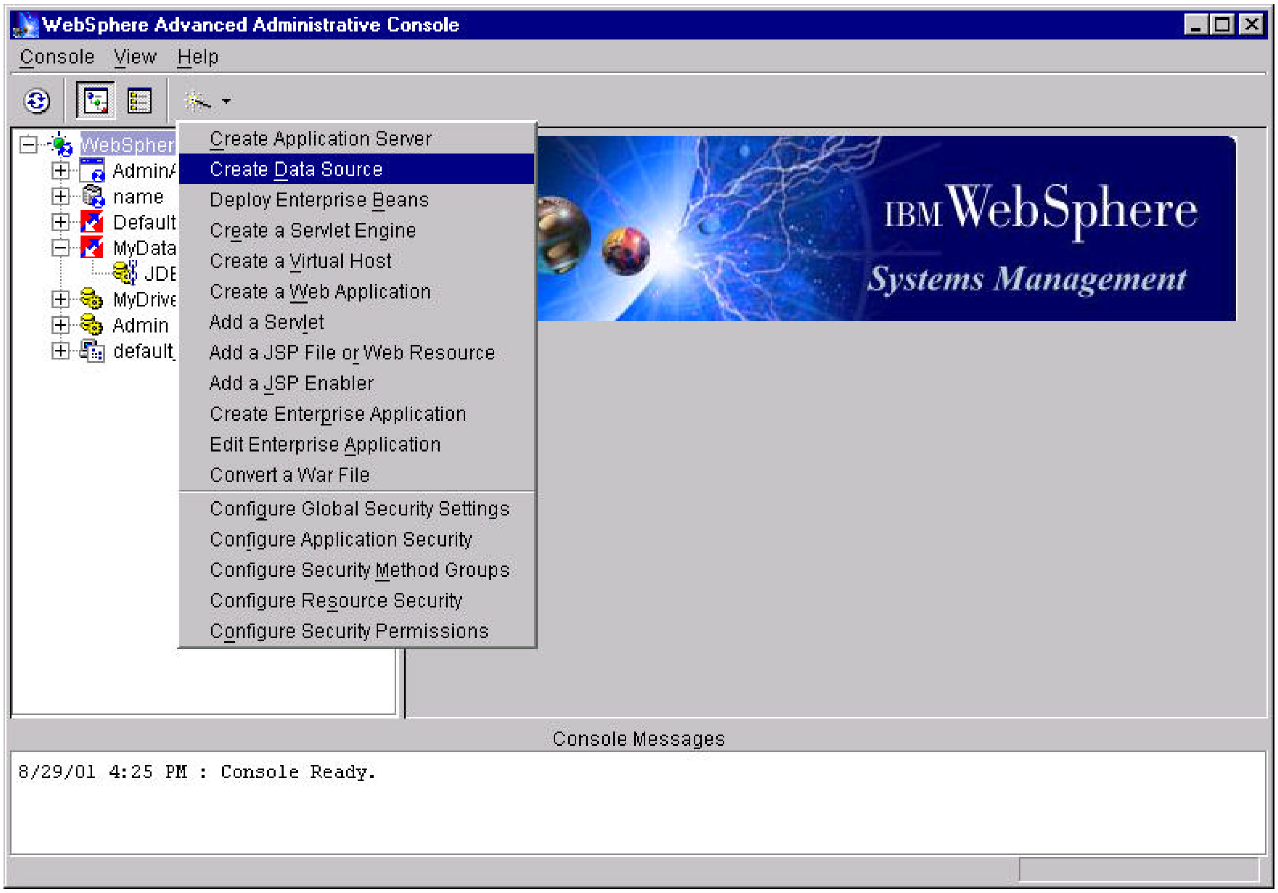
As noted above, several classes will have to be developed on a custom basis for each application, these being the Business Objects and Mappers. See the usage scenario steps for a detailed explanation on how to create these classes. These will also need to be deployed on the WebSphere server.

Finally, a Data Source will have to be created in the WebSphere Administrator's Console and pointed to the database to be used with the framework. The Data Source will require an appropriate JDBC driver for the database that will be used with the framework. The jar file for this driver must be on the WebSphere server, and its classpath should refer to it. The name of the Data Source will be set in the Domain object when the framework is initialized.

6.3.5 Configuration

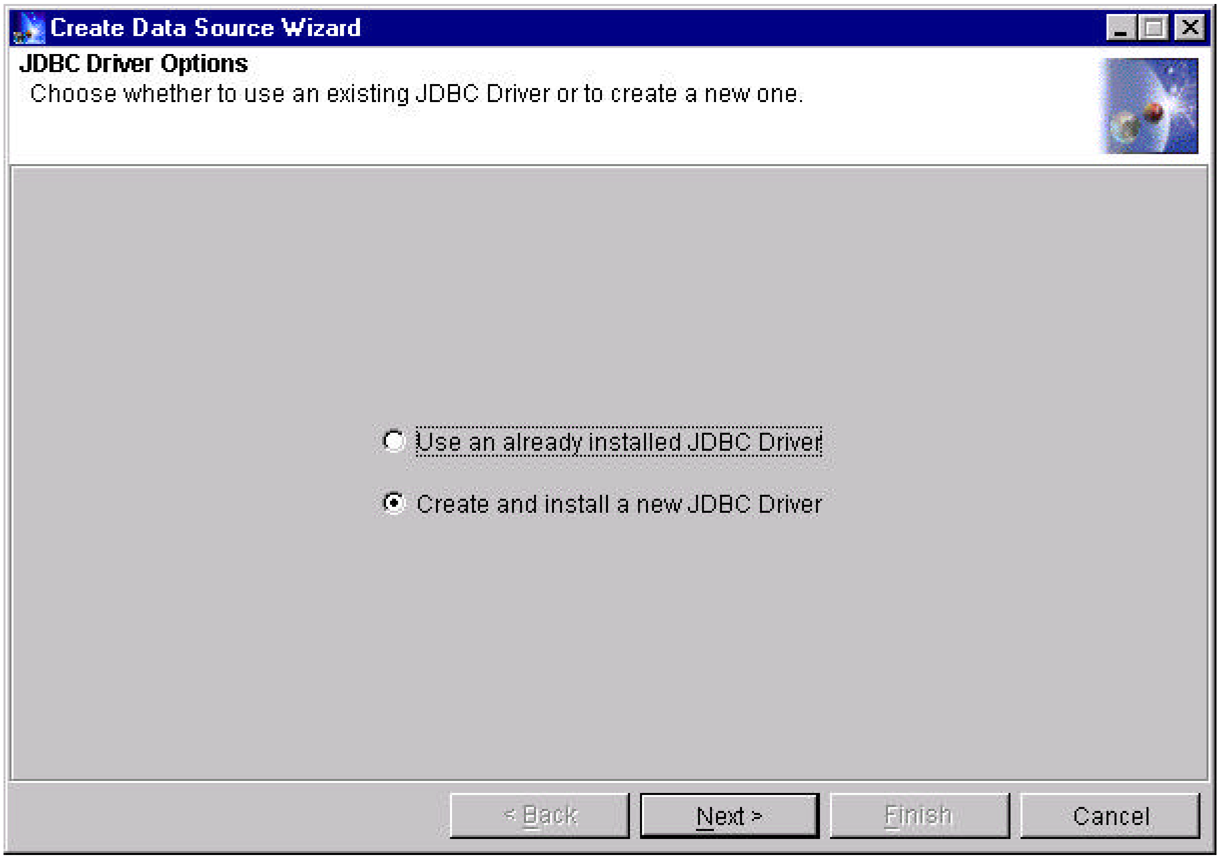
A JDBC 2.0 Data Source must be configured on the WebSphere server in order to access the database that the framework will work with. Note that the Data Source will require an appropriate JDBC driver for the database that will be used with the framework. The jar file for this driver must be on the WebSphere server, and its classpath should refer to it. The following screenshots illustrate how to accomplish this from the WebSphere Administrator's console.

6.3.5.1 Start the Create Data Source Wizard


WebSphere Advanced Administrative Console
Console
View
Help
WebSpher
Admin/
name
Default
MyData
JDE
MyDrive
Admin
default
Create Application Server
Create Data Source
Deploy Enterprise Beans
Create a Servlet Engine
Create a Virtual Host
Create a Web Application
Add a Servlet
Add a JSP File or Web Resource
Add a JSP Enabler
Create Enterprise Application
Edit Enterprise Application
Convert a War File
Configure Global Security Settings
Configure Application Security
Configure Security Method Groups
Configure Resource Security
Configure Security Permissions
IBM WebSphere
Systems Management
Console Messages

```
8/29/01 4:25 PM : Console Ready.
```

6.3.5.2 Identify the JDBC driver for this Data Source



Create Data Source Wizard

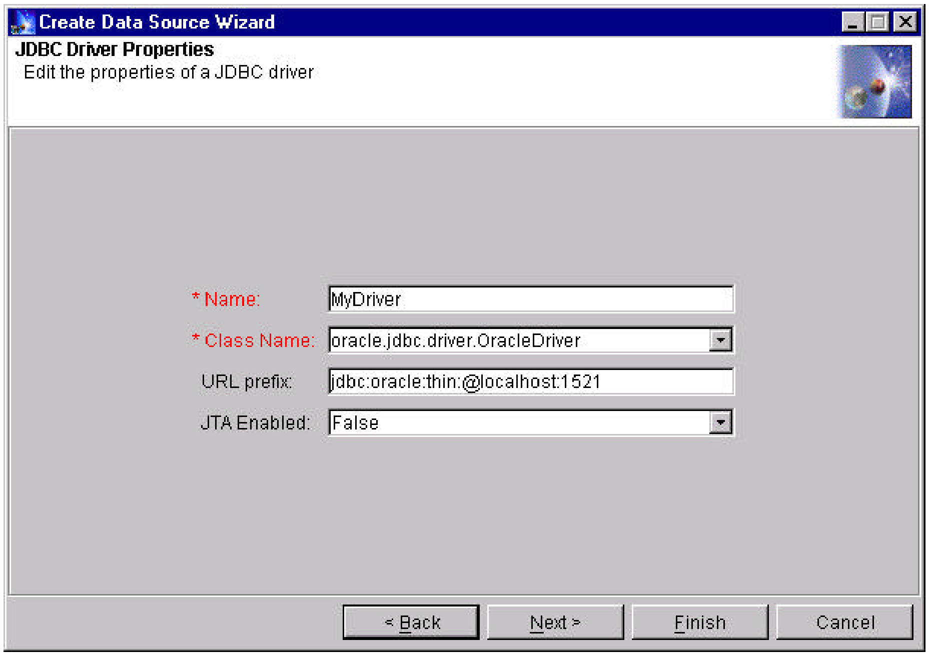
JDBC Driver Options

Choose whether to use an existing JDBC Driver or to create a new one.

○ Use an already installed JDBC Driver

● Create and install a new JDBC Driver

6.3.5.3 Set the properties for the JDBC driver

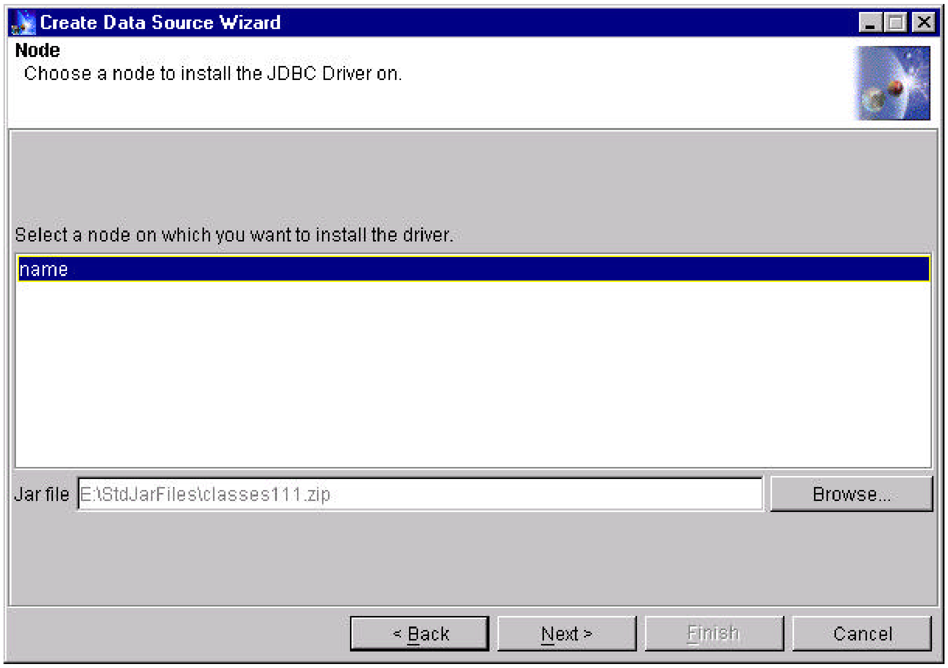


Create Data Source Wizard

JDBC Driver Properties

Edit the properties of a JDBC driver

* Name:	MyDriver
* Class Name:	oracle.jdbc.driver.OracleDriver
URL prefix:	jdbc:oracle:thin:@localhost:1521
JTA Enabled:	False



Create Data Source Wizard

Node

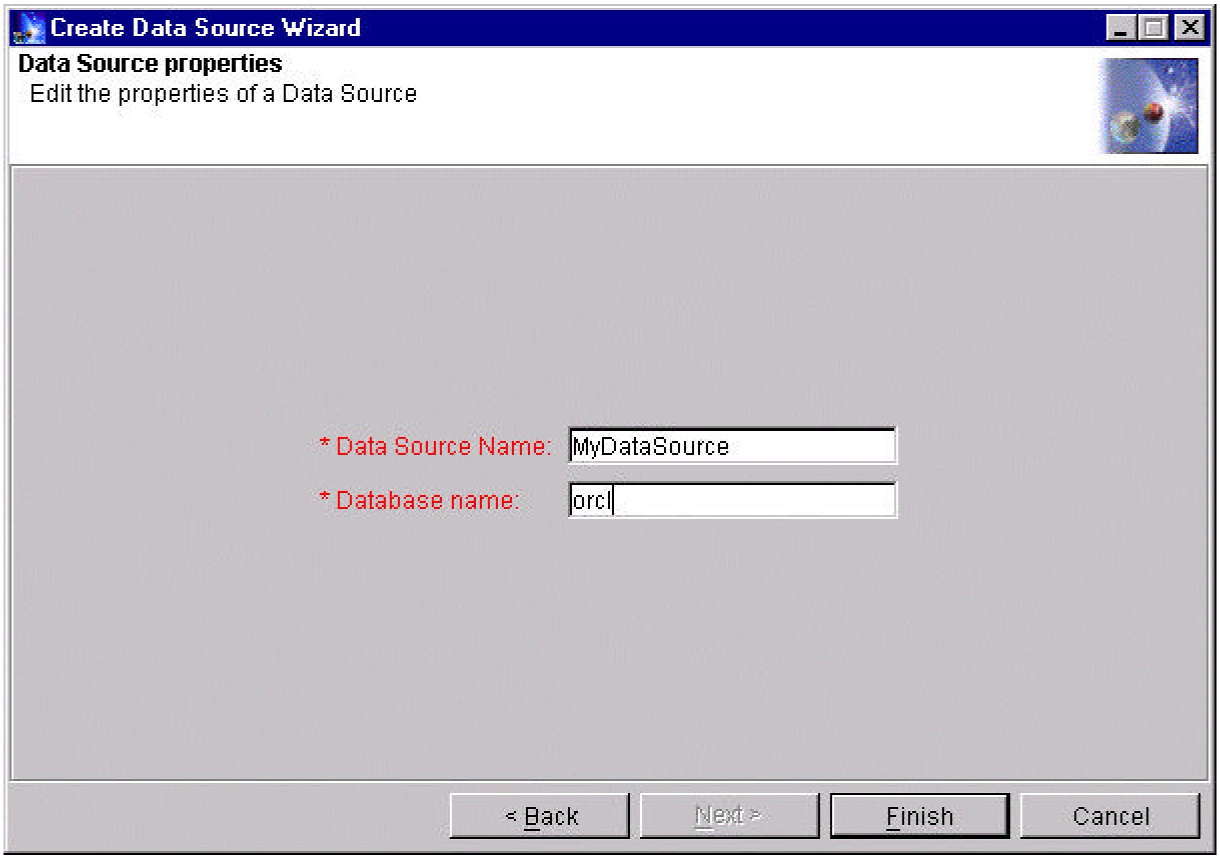
Choose a node to install the JDBC Driver on.

Select a node on which you want to install the driver.

name

Jar file E:\StdJarFiles\classes111.zip

6.3.5.4 Set the properties for the Data Source



Create Data Source Wizard

Data Source properties
Edit the properties of a Data Source

* Data Source Name: MyDataSource

* Database name: orcl

6.3.6 Usage Scenarios and Sample Code

The usage scenarios described in this section describe the steps necessary to completely build and use a sample persistable component. The scenarios encompass all steps in the process, from designing the database tables to retrieving and saving persistable objects in application code. As this sample describes a particular usage in detail, it should also serve to show the power and flexibility of the framework, and how it can be extended to support virtually all interactions with the database during application execution.

6.3.6.1 Design the Database Tables

The tables can be designed using standard database modeling practices – there is nothing about the persistence framework that constrains the database design with which it would work. It should be considered, however, how a business object would map to the table or set of tables in the database. A business object can span multiple tables – for example, a primary table and associated lookup tables for codes, etc. The mapping class will just have to account for the database relationships in its SQL Statements for selects, updates, and so forth.

Below is an example of DDL (simplified, without storage clauses, etc.) for the table used in our sample:

```
CREATE TABLE "SCOTT".T_ORDERREQUEST
(PROPERTYID VARCHAR2(50) NOT NULL,
REQUESTDATE VARCHAR2(50) NULL,
REQUESTACTION VARCHAR2(50) NULL,
EFFECTIVEDATE VARCHAR2(50) NULL,
CHECK (PROPERTYID IS NOT NULL),
UNIQUE  (PROPERTYID),
PRIMARY KEY (PROPERTYID))
PCTFREE 5 PCTUSED 60 INITRANS 2 MAXTRANS 255
STORAGE ( INITIAL 22K NEXT 2K MINEXTENTS 1 MAXEXTENTS 121
PCTINCREASE 0);
```

6.3.6.2 Design and Code the Business Objects

The business objects are the classes that actually hold the data in its programmatic representation. It is what the application developers use to set and retrieve values; when the database is queried or updated, it is these objects that are retrieved or persisted, respectively. They should essentially be implemented as simple beans: private data members for each database field, with public get() and set() methods for each.

The beans can be made more complex with field validation. This is a design decision – the validation can save trips to the database if done properly, but it can also lead to hard-to-maintain code with scattered business rules. However, a well-defined and well-documented set of rules in the business objects will allow for better validation and better performance of the persistence layer without causing debugging and maintainability issues.

Below is a business object that implements a programmatic representation of the database table described above:

```
/**
 * Class Name:    OrderRequestBO
 * Date:            03/05/2001 15:00:33
 * Description:     Persistable Order Request Business Component
 */

public class OrderRequestBO
{
      private String m_propertyid; //The property id associated with
order request
      private String m_requestdate; //The date the request was submitted
      private String m_effectivedate; //The date the request is
effective
      private String m_requestaction; //The action to be performed by
request

      // Constructor
      public OrderRequestBO() {}

      //**************   public getters   ********************/
      /**
      * Get the propertyid attribute
      * @return propertyid
      */
      public String getPropertyid()
      { return this.m_propertyid; }

      /**
      * Get the requestdate attribute
      * @return requestdate
      */
      public String getRequestdate()
      { return this.m_requestdate; }

      /**
      * Get the effectivedate attribute
      * @return effectivedate
      */
      public String getEffectivedate()
      { return this.m_effectivedate; }

      /**
      * Get the requestaction attribute
      * @return requestaction
      */
      public String getRequestaction()
      { return this.m_requestaction; }


      //**************   public setters   *********************/
      /**
      * set the propertyid attribute
      * @param String newPropertyid
      */
      public void setPropertyid(String newPropertyid )
      { this.m_propertyid = newPropertyid; }

      /**
      * set the requestdate attribute
      * @param String newRequestdate
      */
      public void setRequestdate(String newRequestdate )
      { this.m_requestdate = newRequestdate; }

      /**
      * set the effectivedate attribute
      * @param String newEffectivedate
      */
      public void setEffectivedate(String newEffectivedate )
      { this.m_effectivedate = newEffectivedate; }

      /**
      * set the requestaction attribute
      * @param String newRequestaction
      */
      public void setRequestaction(String newRequestaction )
      { this.m_requestaction = newRequestaction; }
}
```

6.3.6.3 Design and Code the mappings between Tables and Objects

The business object mapper is the most critical piece of a persistence framework implementation. It is where the database design meets the Java object design. Therefore, it encapsulates both Java OO concepts and relational database concepts. The example below can appear like a complex component, but it mainly consists of two main functions: 1) parameter retrieval and 2) statement retrieval. It does this for each of the main database interactions: selects, updates, and deletes. Support for these functions is enforced through the ISFAPersistableMapper interface; all mappers much implement it.

The parameter retrieval functions perform the basic task of putting the business object fields into a vector of SFAParameters, with key / value pairs. The key is usually the database field name, and the value is the business object data member that maps to that database field. The parameter also contains the datatype (in the database) for that parameter. The getXXXParameters function simply builds this vector and returns it.

The statement retrieval function returns the properly formatted SQL statement that performs the required operation. The statement has embedded in it placeholders for the parameterized values that the business object hold (and pass to the parameter retrieval function). The placeholders are in the form of ?*parameter_name*? where *parameter_name* is the name given for that field in the parameter retrieval function (again, usually the name of the field in the database works well).

Lastly, the populateAttributeValues() and populateKeyAttributeValues() functions basically tell the mapper which business object to work with – something to consider is that many business objects may be instantiated in code at one point in time (for example, when processing multiple business orders), so these functions serve the purpose of determining the current business object the framework is working with.

Below is an example below shows how these functions are all implemented for the order business object and the table above.

```
/**
 * Class Name:    OrderRequestMapper
 * Date:            03/05/2001 15:00:33
 * Description:     Persistable Order Request Mapping Component
 */

import java.util.Vector;
import gov.ed.sfa.ita.persistence.ISFAPersistableMapper;
import gov.ed.sfa.ita.persistence.SFAResultSet;
import gov.ed.sfa.ita.persistence.SFAParameter;
import gov.ed.sfa.ita.exception.*;

public class OrderRequestMapper implements ISFAPersistableMapper
{
      public OrderRequestBO persistableObject;
      public String m_tableName = "T_ORDERREQUEST";

      public OrderRequestMapper() {
            System.out.println("OrderRequestMapper Instantiated");
      }

      public Vector getDeleteParameters() throws SFAException {
            Vector parameters = new Vector();
            parameters.addElement(new SFAParameter("propertyid",
persistableObject.getPropertyid(),SFAParameter.PT_STRING));
            return parameters;
      }

      public String getDeleteQuery() {
            return "Delete " + " FROM "  +  this.m_tableName + " Where
propertyid = ?propertyid?";
      }

      public Vector getInsertParameters() throws SFAException {
            Vector parameters = new Vector();
            parameters.addElement(new SFAParameter("propertyid",
persistableObject.getPropertyid(),SFAParameter.PT_STRING));
            parameters.addElement(new
SFAParameter("requestdate",persistableObject.getRequestdate(),SFAParam
eter.PT_STRING));
            parameters.addElement(new
SFAParameter("effectivedate",persistableObject.getEffectivedate(),SFAP
arameter.PT_STRING));
            parameters.addElement(new SFAParameter("requestaction",
persistableObject.getRequestaction(),SFAParameter.PT_STRING));
            return parameters;
      }

      public String getInsertQuery() {
            return " INSERT INTO " + this.m_tableName + " (
PROPERTYID, REQUESTDATE, EFFECTIVEDATE, REQUESTACTION)" + " VALUES (
?propertyid? , ?requestdate? , ?effectivedate? , ?requestaction? )" ;
      }

      public Vector getKeySelectParameters() throws SFAException {
            Vector parameters = new Vector();
            parameters.addElement(new SFAParameter("propertyid",
persistableObject.getPropertyid(),SFAParameter.PT_STRING));
            return parameters;
      }

      public String getKeySelectQuery()  {
            return "SELECT propertyid, requestdate, effectivedate,
requestaction" + " FROM "  +  this.m_tableName + " Where propertyid =
?propertyid?" ;
      }

      public String getSelectQuery( String selectCondition) {
            return " SELECT propertyid, requestdate, effectivedate,
requestaction" + " FROM "  +  this.m_tableName + " Where " +
selectCondition ;
      }

      public Vector getUpdateParameters() throws SFAException {
            Vector parameters = new Vector();
            parameters.addElement(new SFAParameter("propertyid",
persistableObject.getPropertyid(),SFAParameter.PT_STRING));
            parameters.addElement(new
SFAParameter("requestdate",persistableObject.getRequestdate(),SFAParam
eter.PT_STRING));
            parameters.addElement(new
SFAParameter("effectivedate",persistableObject.getEffectivedate(),SFAP
arameter.PT_STRING));
            parameters.addElement(new SFAParameter("requestaction",
persistableObject.getRequestaction(),SFAParameter.PT_STRING));
            return parameters;
      }

      public String getUpdateQuery() {
            return " Update " + this.m_tableName + " SET REQUESTDATE =
?requestdate? , EFFECTIVEDATE = ?effectivedate? , REQUESTACTION =
?requestaction?" + " Where PROPERTYID = ?propertyid?" ;
      }
      public Object newFrom(SFAResultSet resultSet) throws
SFAException {
      try {
            OrderRequestBO object = new OrderRequestBO();
            object.setPropertyid((String)
resultSet.getString("propertyid"));
            object.setRequestdate(
resultSet.getString("requestdate"));
            object.setEffectivedate(
resultSet.getString("effectivedate"));
            object.setRequestaction((String)
resultSet.getString("requestaction"));
            return object;
      }
      catch (Exception e) {
            // throw a standard SFAException
            SFAExceptionFactory fac =
SFAExceptionFactory.getInstance();
            SFAException ex = new SFAException();
            Object[] argus = new String[2];
                  argus[0] = "";
                  argus[1] = "";
            Exception prevEx = null;
            ex = fac.createException(SFAException.class, 5, argus,
prevEx, "OrderRequestMapper", "newFrom", "");

            throw ex;
      }
      }

      public void populateAttributeValues(Object obj)  {
            this.persistableObject = (OrderRequestBO) obj;
      }

      public void populateKeyAttributeValues(Object obj)  {
            this.persistableObject = (OrderRequestBO) obj;
      }

      public void setTableName(String tableName) {
            this.m_tableName = tableName;
      }
}
```

6.3.6.4 Set up the database connection

Setting the database connection consists of creating a domain object with a username, password, and DataSource name, and then creating a persistable object manager with that domain.

```
try {
            // Setup and populate connection values
            String dsn = "jdbc/MyDataSource";
            String uid = "scott";
            String pwd = "tiger";
            int inPropertyId = 1234567;
```

```
                    SFADomain domain = new SFADomain(dsn, uid, pwd);
                    SFAPersistableObjectManager pom = new
SFAPersistableObjectManager(domain);
          }
          catch (SFAException ex) {
                    ex.printStackTrace();
                    out.println("error " + ex.getOrigException().getMessage());
          }
```

6.3.6.5 Use the Framework to retrieve Business Objects

Retrieving a business object from the persistence framework involves creation of an empty business object of that type, setting its key values so the framework knows which object to retrieve, and requesting the object via the getObject() method of the persistable object manger.

```
try {
            // Setup and populate connection values
            String dsn = "jdbc/MyDataSource";
            String uid = "scott";
            String pwd = "tiger";
            int inPropertyId = 1234567;

            SFADomain domain = new SFADomain(dsn, uid, pwd);
            SFAPersistableObjectManager pom = new
SFAPersistableObjectManager(domain);
            OrderRequestMapper mapper =  new OrderRequestMapper();

            OrderRequestBO object = new OrderRequestBO();
            object.setPropertyid(String.valueOf(inPropertyId));
            mapper.populateAttributeValues(object);

            object = (OrderRequestBO) pom.getObject(mapper,
      mapper.getKeySelectQuery(), mapper.getKeySelectParameters());
            out.println("Retrieved fields for this object: <br>");
            out.println(object2.getPropertyid() + "<br>");
            out.println(object2.getEffectivedate() + "<br>");
            out.println(object2.getRequestdate() + "<br>");


      }
      catch (SFAException ex) {
            ex.printStackTrace();
            out.println("error " +
ex.getOrigException().getMessage());
      }
```

6.3.6.6 Update properties of Business Objects

Updating the database with a persistable business object through the persistence framework first requires the creation of a business object of that type. This object could have been created using the *new* Java keyword, or retrieved from the persistent store using the getObject method of the persistable object manager (see previous section). Once created, the object's properties are updated using the standard set() methods.

```
try {
            // Setup and populate connection values
            String dsn = "jdbc/MyDataSource";
            String uid = "scott";
            String pwd = "tiger";
            int inPropertyId = 1234567;

            SFADomain domain = new SFADomain(dsn, uid, pwd);
            SFAPersistableObjectManager pom = new
SFAPersistableObjectManager(domain);
            OrderRequestMapper mapper =  new OrderRequestMapper();

            OrderRequestBO object = new OrderRequestBO();
            object.setPropertyid(String.valueOf(inPropertyId));
            mapper.populateAttributeValues(object);

            object = (OrderRequestBO) pom.getObject(mapper,
      mapper.getKeySelectQuery(), mapper.getKeySelectParameters());
            out.println("Retrieved fields for this object: <br>");
            out.println(object2.getPropertyid() + "<br>");
            out.println(object2.getEffectivedate() + "<br>");
            out.println(object2.getRequestdate() + "<br>");

            /* Set the new values for the fields for this object */

            String inDateRequested = "01/01/01";
            String inDateEffective = "01/01/01";
            String inActionRequested = "Place Order";

            object.setEffectivedate(inDateEffective);
            object.setRequestaction(inActionRequested);
            object.setRequestdate(inDateRequested);


      }
      catch (SFAException ex) {
            ex.printStackTrace();
            out.println("error " +
ex.getOrigException().getMessage());

               }
```

6.3.6.7 Use the Framework to persist Business Objects to the database

With the key values set (so the framework knows which object to update) along with any additional data members that need to be updated (see previous section), the object can be persisted via the updateObject() method of the persistable object manger.

```
      try {
            // Setup and populate connection values
            String dsn = "jdbc/MyDataSource";
            String uid = "scott";
            String pwd = "tiger";
            int inPropertyId = 1234567;

            SFADomain domain = new SFADomain(dsn, uid, pwd);
            SFAPersistableObjectManager pom = new
SFAPersistableObjectManager(domain);
            OrderRequestMapper mapper =  new OrderRequestMapper();

            OrderRequestBO object = new OrderRequestBO();
            object.setPropertyid(String.valueOf(inPropertyId));
            mapper.populateAttributeValues(object);

            object = (OrderRequestBO) pom.getObject(mapper,
      mapper.getKeySelectQuery(), mapper.getKeySelectParameters());
            out.println("Retrieved fields for this object: <br>");
            out.println(object2.getPropertyid() + "<br>");
            out.println(object2.getEffectivedate() + "<br>");
            out.println(object2.getRequestdate() + "<br>");

            /* Set the new values for the fields for this object */

            String inDateRequested = "01/01/01";
            String inDateEffective = "01/01/01";
            String inActionRequested = "Place Order";

            object.setEffectivedate(inDateEffective);
            object.setRequestaction(inActionRequested);
            object.setRequestdate(inDateRequested);

            /* Persist the object out to the database */
            mapper.populateAttributeValues(object);

      pom.addObject(mapper.getInsertQuery(),mapper.getInsertParameters());

            pom.commitTransaction();

      }
      catch (SFAException ex) {
            ex.printStackTrace();
            out.println("error " + ex.getOrigException().getMessage());
      }
```

6.3.6.8 Obtain a pooled database connection from the Framework for ad-hoc JDBC queries

The persistable object manager also contains a method that allows an application programmer to directly use the JDBC connection that the framework is using. This may prove useful if the application development decides that it needs to perform some database access outside of the persistence framework. A developer would have to be knowledgeable in databases, SQL, and the JDBC API in order to make use of the connection. Use of the connection is only advised if a database access issue cannot fit within the design of the framework, which shouldn't be often, but if it occurs, then the framework will provide the pooled connection, created using the best practices for access specified for IBM WebSphere.

```
try {
      Connection connection = pom.getConnection();

      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM EMP");

      out.println("<table border=1>");
      int i = 0;
      while (rs.next()) {
            out.println("<tr><td>" + rs.getString("EMPNO") + "</td><td>"
+ rs.getString("ENAME") + "</td><td>"  + rs.getString("JOB") +
"</td></tr>");
            i++;
      }
      out.println("</table>");
      rs.close();
      stmt.close();
      connection.close();
      }
      catch (SFAException ex) {
            ex.printStackTrace();
            out.println("error " + ex.getOrigException().getMessage());
      }
```

6.3.7 Resources

JavaDoc on JDBC - http://www.javasoft.com/products/jdk/1.2/docs/api/index.html

JDBC Tutorial - http://www.javasoft.com/docs/books/tutorial/jdbc/index.html

7 RCS – Search Framework

The search framework simplifies, standardizes, and improves the use of the Autonomy search engine. The Autonomy Search Engine server provides a variety of methods to utilize its search capabilities including a C API, HTTP API, and configurable CGI program. Of the three ITA R1.0 applications that used Autonomy, IFAP and Intranet 2.0 used the C API to write a custom CGI program and Schools Portal used the configurable CGI program. Neither of these applications followed the J2EE standards of the ITA environment nor did the applications utilize all of the features provided by the Autonomy Search Engine Server.

The framework consists of classes that provide a common way to access the Autonomy HTTP API and utilize the following Autonomy features:

- Query search engine (including querying a custom field)
- Natural Language or "Fuzzy" query search engine (including querying a custom field)
- Display search results (including sorting by a custom field)
- Suggest additional search results

7.1 Testing Conditions & Results

7.1.1 Automated Testing

Cycle 1 – Autonomy statement

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	customQuery Stmt is null	TestAutonomyStatement	testAddCustomQueryStmtErr	Autonomy Statement	addCustomQueryStmt	Custom Query is not executed and an SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
2	customQuery Stmt is not null	TestAutonomyStatement	testAddCustomQueryStmt	Autonomy Statement	addCustomQueryStmt	String is added to end of query	autonomy.properties, rcs.xml, errormessages.properties
3	The max number of results for the fuzzy query is not set	TestAutonomyStatement	testExecuteFuzzyQueryErr1	Autonomy Statement	executeFuzzyQuery	The fuzzy query is not executed and an SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
4	The query text or custom query text is not set	TestAutonomyStatement	testExecuteFuzzyQueryErr2	Autonomy Statement	executeFuzzyQuery	The fuzzy query is not executed and an SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
5	The search criteria is met	TestAutonomyStatement	testExecuteFuzzyQuery	Autonomy Statement	executeFuzzyQuery	The fuzzy query is executed	autonomy.properties, rcs.xml, errormessages.properties
6	The max number of results for the query is not set	TestAutonomyStatement	testExecuteQueryErr1	Autonomy Statement	executeQuery	The query is not executed and an SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
7	The query text or custom query text is not set	TestAutonomyStatement	testExecuteQueryErr2	Autonomy Statement	executeQuery	The query is not executed and an SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
8	The criteria for the query is met	TestAutonomyStatement	testExecuteQuery	Autonomy Statement	executeQuery	The query is executed	autonomy.properties, rcs.xml, errormessages.properties
9	The docid is invalid	TestAutonomyStatement	testExecuteSuggestErr1	Autonomy Statement	executeSuggest	The Suggest query is not executed, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
10	The max number of results for the Suggest is 0	TestAutonomyStatement	testExecuteSuggestErr2	Autonomy Statement	executeSuggest	The Suggest query is not executed, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
11	All the Suggest query criteria are met	TestAutonomyStatement	testExecuteSuggest	Autonomy Statement	executeSuggest	The Suggest query is executed	autonomy.properties, rcs.xml, errormessages.properties
12	The query or search starts over	TestAutonomyStatement	testReset	Autonomy Statement	reset	The query terms are all reset to their initial values	autonomy.properties, rcs.xml, errormessages.properties
13	The inputted database name is not valid	TestAutonomyStatement	testSetDatabaseNamesErr	Autonomy Statement	setDatabaseNames	Database name not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
14	The inputted database name is valid	TestAutonomyStatement	testSetDatabaseNames	Autonomy Statement	setDatabaseNames	The inputted database name is concatenated to the database name string	autonomy.properties, rcs.xml, errormessages.properties
15	The inputted database number is not valid	TestAutonomyStatement	testSetDatabaseNumsErr	Autonomy Statement	setDatabaseNums	The database number is not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
16	The inputted database number is valid	TestAutonomyStatement	testSetDatabaseNums	Autonomy Statement	setDatabaseNums	The inputted database number is set	autonomy.properties, rcs.xml, errormessages.properties
17	The docID is not null	TestAutonomyStatement	testSetID	Autonomy Statement	setID	The document ID is set for the Suggestion query	autonomy.properties, rcs.xml, errormessages.properties
18	The docID is null	TestAutonomyStatement	testSetIDErr	Autonomy Statement	setID	The ID is not set, SFAException throw	autonomy.properties, rcs.xml, errormessages.properties
19	A valid number of max results is inputted	TestAutonomyStatement	testSetMaxNumResults	Autonomy Statement	setMaxNumResults	The MaxNumResults is set	autonomy.properties, rcs.xml, errormessages.properties
20	Max number of results set to < 0	TestAutonomyStatement	testSetMaxNumResultsErr	Autonomy Statement	setMaxNumResults	MaxNumResults not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
21	A zero or positive number is inputted for threshold	TestAutonomyStatement	testSetMinThreshold	Autonomy Statement	setMinThreshold	The threshold is set	autonomy.properties, rcs.xml, errormessages.properties

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
22	A negative number is inputted for the threshold	TestAutonomyStatement	testSetMinThresholdErr	Autonomy Statement	setMinThreshold	Threshold is not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
23	boolean AND search is selected	TestAutonomyStatement	testSetQueryAllWords	Autonomy Statement	setQueryAllWords	The search is set to find all of the words	autonomy.properties, rcs.xml, errormessages.properties
24	boolean OR search is selected	TestAutonomyStatement	testSetQueryAnyWords	Autonomy Statement	setQueryAnyWords	The search is set to find any of the words	autonomy.properties, rcs.xml, errormessages.properties
25	The length of the query string is at least one character	TestAutonomyStatement	testSetQueryText	Autonomy Statement	setQueryTest	The query text is set	autonomy.properties, rcs.xml, errormessages.properties
26	The query text is null	TestAutonomyStatement	testSetQueryTextErr	Autonomy Statement	setQueryTest	Query text is not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
27	Field name and query text are valid	TestAutonomyStatement	testSetQueryTextCustomField	Autonomy Statement	setQueryTextCustomField	Custom query text is set	autonomy.properties, rcs.xml, errormessages.properties
28	Field name is null for custom field query	TestAutonomyStatement	testSetQueryTextCustomFieldErr1	Autonomy Statement	setQueryTextCustomField	Custom query text is not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
29	Custom field is null for custom field query	TestAutonomyStatement	testSetQueryTextCustomFieldErr2	Autonomy Statement	setQueryTextCustomField	Custom query text is not set, SFAException thrown	autonomy.properties, rcs.xml, errormessages.properties
30	Sorting by date is requested	TestAutonomyStatement	testSetSortByDate	Autonomy Statement	setSortByDate	The query to return results sorted by date only	autonomy.properties, rcs.xml, errormessages.properties
31	Sorting by relevance is requested	TestAutonomyStatement	testSetSortByRelevance	Autonomy Statement	setSortByRelevance	The query to return results sorted by relevance only	autonomy.properties, rcs.xml, errormessages.properties
32	Sorting by relevance date is requested	TestAutonomyStatement	testSetSortByRelevanceDate	Autonomy Statement	setSortByRelevanceDate	The query to return results sorted by relevance date only	autonomy.properties, rcs.xml, errormessages.properties
33	Return autonomy resultset from statement	TestAutonomyStatement	testGetRS	Autonomy Statement	getRS	A reference to the autonomy resultset is returned	autonomy.properties, rcs.xml, errormessages.properties

Cycle 2 – Autonomy result set

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Valid number of rows are specified	TestAutonomyResultSet	testAbsoluteValidRows	AutonomyResultSet	absolute	Jump to specified row	autonomy.properties, rcs.xml, errormessages.properties
2	Invalid number of rows are specified	TestAutonomyResultSet	testAbsoluteInvalidRows	AutonomyResultSet	absolute	Does not jump to specified row	autonomy.properties, rcs.xml, errormessages.properties
3	An invalid fieldname is passed	TestAutonomyResultSet	testGetCustomFieldInvalid	AutonomyResultSet	getCustomField	An SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
4	A valid fieldname is passed	TestAutonomyResultSet	testGetCustomField	AutonomyResultSet	getCustomField	Returns a parsed result set	autonomy.properties, rcs.xml, errormessages.properties
5	The result set contains a valid row	TestAutonomyResultSet	testGetDatabaseNum	AutonomyResultSet	getDatabaseNum	Returns Autonomy database number of current row	autonomy.properties, rcs.xml, errormessages.properties
6	The result set contains a valid row	TestAutonomyResultSet	testGetID	AutonomyResultSet	getID	Returns document ID of current row	autonomy.properties, rcs.xml, errormessages.properties
7	A valid result set exists	TestAutonomyResultSet	testGetNumResults	AutonomyResultSet	getNumResults	Returns number of results from result set	autonomy.properties, rcs.xml, errormessages.properties
8	The result set contains a valid row	TestAutonomyResultSet	testGetQuickSummary	AutonomyResultSet	getQuickSummary	Returns a quick summary of current row	autonomy.properties, rcs.xml, errormessages.properties
9	The result set contains a valid row	TestAutonomyResultSet	testGetRow	AutonomyResultSet	getRow	Returns row number	autonomy.properties, rcs.xml, errormessages.properties
10	The result set contains a valid row	TestAutonomyResultSet	testGetSummary	AutonomyResultSet	getSummary	Returns a summary of current row	autonomy.properties, rcs.xml, errormessages.properties
11	The result set contains a valid row	TestAutonomyResultSet	testGetTitle	AutonomyResultSet	getTitle	Returns title of current row	autonomy.properties, rcs.xml, errormessages.properties
12	The result set contains a valid row	TestAutonomyResultSet	testGetURL	AutonomyResultSet	getURL	Returns URL of current row	autonomy.properties, rcs.xml, errormessages.properties
13	The result set contains a valid row	TestAutonomyResultSet	testGetWeight	AutonomyResultSet	getWeight	Returns document weight of current row	autonomy.properties, rcs.xml, errormessages.properties
14	The current row isn't the last row	TestAutonomyResultSet	testNextNotLastRow	AutonomyResultSet	next	The next row is set and true is returned	autonomy.properties, rcs.xml, errormessages.properties
15	The current row is the last row	TestAutonomyResultSet	testNextIsLastRow	AutonomyResultSet	next	The current row is set and failure is returned	autonomy.properties, rcs.xml, errormessages.properties
16	The current row isn't the first row	TestAutonomyResultSet	testPreviousIsNotFirstRow	AutonomyResultSet	previous	The previous row is set and true is returned	autonomy.properties, rcs.xml, errormessages.properties
17	The current row is the first row	TestAutonomyResultSet	testPreviousIsFirstRow	AutonomyResultSet	previous	The current row is set and false is returned	autonomy.properties, rcs.xml, errormessages.properties
18	A valid result set	TestAutonomy	testGetNumRows	AutonomyRe	getNumRo	Returns	autonomy.properties, rcs.xml,

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
	exists	ResultSet		sultSet	ws	number of ows in result set	errormessages.properties

Cycle 3 – Autonomy server connection up

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	AutonomyConnection is created	TestAutonomyConnectionServerDown	TestAutonomyConnectionConstrPass	AutonomyConnection	AutonomyConnection	AutonomyConnection is created	autonomy.properties, rcs.xml, errormessages.properties
2	An AutonomyStatement is created	TestAutonomyConnectionServerUp	testCreateStatementPass	AutonomyConnection	createStatement	AutonomyStatement is created	autonomy.properties, rcs.xml, errormessages.properties
3	Host name is set to valid Autonomy Server host name	TestAutonomyConnectionServerUp	testGetHost	AutonomyConnection	getHost	Host name returned	autonomy.properties, rcs.xml, errormessages.properties
4	Index port is set to valid Autonomy Server index port	TestAutonomyConnectionServerUp	testGetIndexPort	AutonomyConnection	getIndexPort	index port is returned	autonomy.properties, rcs.xml, errormessages.properties
5	Query port is set to valid Autonomy Server query port	TestAutonomyConnectionServerUp	testGetQueryPort	AutonomyConnection	getQueryPort	query port is returned	autonomy.properties, rcs.xml, errormessages.properties
6	The server is up	TestAutonomyConnectionServerUp	testIsServerStatusOkTrue	AutonomyConnection	isServerStatusOK	returns true	autonomy.properties, rcs.xml, errormessages.properties

Cycle 4 – Autonomy server connection down

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	AutonomyConnection not created, server is down	TestAutonomyConnectionServerDown	TestAutonomyConnectionConstrFail	AutonomyConnection	AutonomyConnection	AutonomyConnection is not created, SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
2	An AutonomyStatement is not made, server is down	TestAutonomyConnectionServerDown	testCreateStatementFail	AutonomyConnection	createStatement	AutonomyStatement is not created, SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties
3	The server is down	TestAutonomyConnectionServerDown	testIsServerStatusOkFalse	AutonomyConnection	isServerStatusOK	Returns false	autonomy.properties, rcs.xml, errormessages.properties

Cycle 5 – Autonomy connection bad prop.

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Input properties file is not found	TestAutonomyConnectionBadProps	TestAutonomyConnectionBadPropsFile	Autonomy Connection	AutonomyConnection	AutonomyConnection is not created, SFAException is thrown	autonomy.properties, rcs.xml, errormessages.properties

7.2 Performance Analysis

7.2.1 Summary

The ITA Search framework was run through a performance testing harness to determine its system-level performance metrics. The ITA Search framework has strong performance characteristics. Its tiered architecture of connections, statements and resultsets ensures that the only processing that occurs is what is absolutely necessary for an operation.

The Search framework does make use of the Logging framework. Logging includes disk accesses (generally very time-costing operations) and thus in a production environment where logging is kept to a minimum, the response times should be faster. The actual response of the Autonomy Server is very fast. Network latency enters into the equation somewhat, but not significant. Nevertheless, The most important enhancement may be made to the parsing of results. It has been discovered that there is an increase in response time when the number of results increase. This is due to the parsing the results after it is retrieved from the Autonomy Server. As indicated by the *Methods with the most time spent only in that method* chart shown below, the search framework spends the most amount of time parsing the search result. The upcoming release of the Search framework will include revised code to reduce the time spend on parsing.

7.2.2 Test Environment

The testing harness was run on a standard SFA developer workstation. The hardware consisted of a Compaq Deskpro with a single 600 MHz Pentium III processor and 512 MB of RAM. The machine ran Windows NT 4.0 Service Pack 6. The Java environment was Sun's JDK 1.3. While tests were run, no other applications were loaded into memory, and the system was not interacted with. This was done in order to leave all resources available to the test harness, and eliminate the possibility of unexplained behavior in the tables of results.

7.2.3 Test Configuration

The ITA Search framework was configured in a very standard manner, as it would be in actual usage. The configuration of the framework implements Logging and Exception frameworks. Therefore, any configurations these frameworks require were used. To eliminate network latency the same workstation was used for an Autonomy Server instead of traversing through the frame relay network. This should simulate the same characteristics of a production environment where the application server has a very fast connection to the Autonomy server.

7.2.4 Test Scenarios

This search performance test focused on one usage scenario for its analysis: Looping 25 times through a method invoking AutonomyConnection, AutonomyStatement, and AutonomyResultSet. ITA team chose to consider 25 sequential runs to reach a steady state of object use. Reaching a steady-state will show that memory allocations are consistent. This scenario covers all three components of the Search framework. The first two components were called 25 times each, while the third (resultset) component was called 250 times (because we set a maximum of 10 results).

Here is the testing code:

```
package gov.ed.sfa.ita.samples;

import gov.ed.sfa.ita.exception.*;
import gov.ed.sfa.ita.search.*;
import gov.ed.sfa.ita.logging.*;

public class AutonomyPerform {

   public AutonomyPerform() {
      super();
   }

   public static void main(String[] args) {
            for (int x=0; x<25; x++) {
                  testSearchNormal();
            }
   }

   public static boolean testSearchNormal() {
      try {
         AutonomyConnection con = new AutonomyConnection();
         con.getHost();
         con.getIndexPort();
         con.getQueryPort();

         AutonomyStatement stmt = con.createStatement();

         stmt.setDatabaseNums("34");
         stmt.setMaxNumResults(10);
         stmt.setMinThreshold(10);
         stmt.setQueryText("SFA Publications");
         stmt.getQueryText();

         AutonomyResultSet aRS = stmt.executeQuery();

         aRS.getNumResults();
         do {
            aRS.getRow();
            aRS.getNumResults();
            aRS.getTitle();
            aRS.getSummary();
            aRS.getQuickSummary();
            aRS.getURL();
            aRS.getWeight();
            aRS.getID();
            aRS.getDatabaseNum();
            aRS.getCustomField("Posted_Date");
         }
         while (aRS.next());
      }
      catch (SFAException e) {
         System.out.println(e.getMessage());
         return false;
      }

      return true;
   }

}
```

7.2.5 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.

2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.

3. Code Efficiency: To identify any performance bottleneck due to inefficient code algorithms

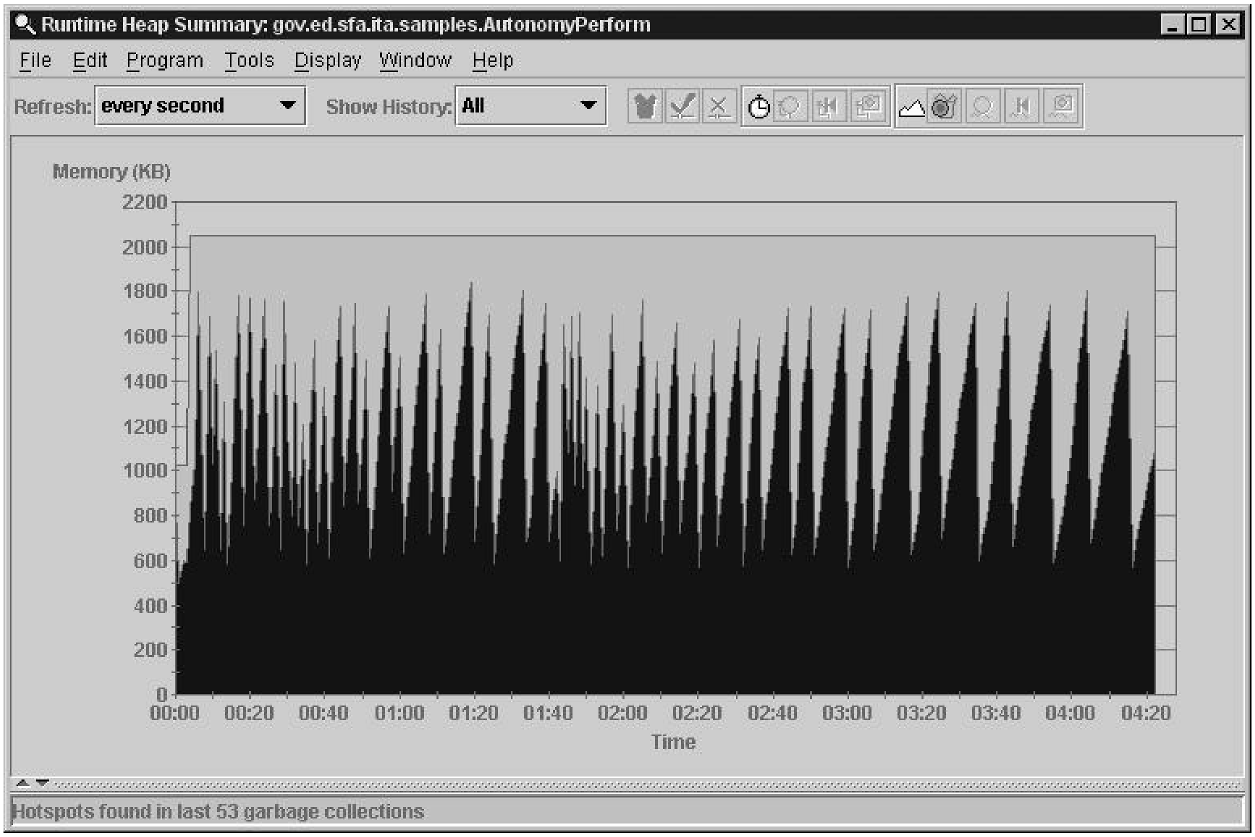
7.2.5.1 Memory (Heap) Usage

The performance test utilized JProbe Profiler's Memory Debugger to identify the parts of the framework that might be causing loitering objects. This was accomplished by analysis of the Java heap. The Runtime Heap Summary window can be used to view instance counts and information on allocating methods.

The Heap Usage Chart below plots the size of the Java heap at a time interval of 1 second. The chart helps to visualize memory use in the Java heap. It displays the available size of the Java heap (the light gray line above 2000 KB) and the used memory (the dark lines with peaks) over time.

The heap usage chart shows a series of spikes. Steep spikes in the Heap Usage Chart represent temporary objects being allocated and garbage collected. If the levels of the troughs become higher over time, then not all the temporary objects are garbage collected. As can be seen the troughs remain steady over time, and while multiple objects are being created and destroyed, there do not appear to be any lingering objects.

Runtime Heap Usage



7.2.5.2 Garbage Collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage-collected objects accumulates. The list below is the final top ten, displaying cumulative objects created at the end of program execution.

Each row identifies the class by package name, if any, and class name. The next columns state, in order, the number of garbage collected objects (GC'ed column) for the class, the number of instances remaining in the heap (Alive column), and the method that allocated the instances of the class (AllocatedAt column). The same class can appear more than once because more than one method allocated instances of the class.

The chart below does not show any unexpected activity, or activity that would indicate a performance problem. Most of the objects created are strings, string buffers, or character arrays. These numbers are in line with the framework requirements and expected behavior.

Garbage Collection Statistics

Package	Class	GC'ed	Alive	Allocated At
	char[]	143,637	1,594	String.<init>
java.lang	String	138,833	1,566	BufferedReader.readLine
java.lang	String	135,890	1,630	Stringsubstring
java.util	StringTokenizer	134,195	1,555	AutonomyResultSet.parseResult
java.lang	String	9,472	130	StringBuffer.toString
	char[]	7,503	92	StringBuffer.<init>
	char[]	7,281	49	String.<init>
com.protomatter.syslog	SyslogMessage	7,257	45	Syslog.log
	Object[]	7,257	45	Syslog.log
java.lang	String	7,257	45	String.valueOf

7.2.5.3 Code Efficiency

There are nine efficiency metrics that can be collected in JProbe — five basic metrics and four compound metrics. The basic metrics include Number of Calls, Method Time, Cumulative Time, Method Object Count, and Cumulative Object Count. The compound metrics are averages per number of calls, including Average Method Time, Average Cumulative Time, Average Method Object Count, and Average Cumulative Object Count. Time is measured as cpu time.

The following list defines the nine performance metrics:

- Number of Calls - The number of times the method was invoked.
- Method Time - The amount of time spent executing the method, excluding time spent in its descendants.
- Cumulative Time - The total amount of time spent executing the method, including time spent in its descendants but excluding time spent in recursive calls to descendants.
- Method Object Count - The number of objects created during the method’s execution, excluding those created by its descendants.
- Cumulative Object Count - The total number of objects created during the method’s execution, including those created by its descendants.
- Average Method Time - Method Time divided by Number of Calls.
- Average Cumulative Time - Cumulative Time divided by Number of Calls.
- Average Method Object - Count Method Object Count divided by Number of Calls.

The charts on the following pages serve to document the performance characteristics of the search framework with lists based on the above metrics:

- Number of Calls
- Cumulative Time
- Method Time
- Average Cumulative Time
- Average Method Time

These measures are basic indicators of processing resource utilization. The lists can be reviewed for unexpected activity or optimization opportunities. (Note: all times are in milliseconds and all data object sizes are in kilobytes)

Methods with the most calls:

Number of Calls

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
BufferedReader.readLine()	140,625	95,887.04 (40.0%)	95,887.04 (40.0%)	0.68 (0.0%)	0.68 (0.0%)	280,728 (42.5%)	1 (0.0%)
StringTokenizer.nextToken()	137,350	49,706.93 (20.7%)	49,706.93 (20.7%)	0.36 (0.0%)	0.36 (0.0%)	137,826 (20.9%)	1 (0.0%)
StringTokenizer.hasMoreTokens()	135,925	1,014.84 (0.4%)	1,014.84 (0.4%)	0.01 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)
StringTokenizer.<init>(String, String)	135,775	2,743.07 (1.1%)	2,743.07 (1.1%)	0.02 (0.0%)	0.02 (0.0%)	0 (0.0%)	0 (0.0%)
String.equals(Object)	135,550	763.64 (0.3%)	763.64 (0.3%)	0.01 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)
StringBuffer.append(String)	13,050	1,065.08 (0.4%)	1,065.08 (0.4%)	0.08 (0.0%)	0.08 (0.0%)	2,300 (0.3%)	0 (0.0%)
StringBuffer.toString()	8,451	3,366.04 (1.4%)	3,366.04 (1.4%)	0.40 (0.0%)	0.40 (0.0%)	8,451 (1.3%)	1 (0.0%)
Syslog.log(Object, Object, Object, Object, int)	7,302	8,771.81 (3.7%)	60.29 (0.0%)	1.20 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)
Syslog.log(InetAddress, Object, Object, Object, Object, int)	7,302	8,711.52 (3.6%)	8,711.52 (3.6%)	1.19 (0.0%)	1.19 (0.0%)	29,284 (4.4%)	4 (0.0%)
StringBuffer.<init>(String)	6,326	1,999.53 (0.8%)	1,999.53 (0.8%)	0.32 (0.0%)	0.32 (0.0%)	6,326 (1.0%)	1 (0.0%)
StringBuffer.append(int)	4,225	4,923.47 (2.1%)	4,923.47 (2.1%)	1.17 (0.0%)	1.17 (0.0%)	12,925 (2.0%)	3 (0.0%)
BUFFEREDREADER.<INIT>(READER)	2,050	8,972.77 (3.7%)	8,972.77 (3.7%)	4.38 (0.0%)	4.38 (0.0%)	2,050 (0.3%)	1 (0.0%)
BufferedReader.close()	2,025	30.14 (0.0%)	30.14 (0.0%)	0.01 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
StringReader.<init>(String)	2,025	10.05 (0.0%)	10.05 (0.0%)	0.00 (0.0%)	0.00 (0.0%)	0 (0.0%)	0 (0.0%)
AutonomyResultSet.parseResult(String, int)	2,000	218,190.00 (91.1%)	50,470.57 (21.1%)	109.09 (0.0%)	25.24 (0.0%)	143,758 (21.8%)	71 (0.0%)

Methods with the most total time (includes time spent in sub-methods):

Cumulative Time

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
AutonomyResultSet.parseResult(String, int)	2,000	218,190.00 (91.1%)	50,470.57 (21.1%)	109.09 (0.0%)	25.24 (0.0%)	143,758 (21.8%)	71 (0.0%)
BufferedReader.readLine()	140,625	95,887.04 (40.0%)	95,887.04 (40.0%)	0.68 (0.0%)	0.68 (0.0%)	280,728 (42.5%)	1 (0.0%)
StringTokenizer.nextToken()	137,350	49,706.93 (20.7%)	49,706.93 (20.7%)	0.36 (0.0%)	0.36 (0.0%)	137,826 (20.9%)	1 (0.0%)
AutonomyResultSet.getCustomField(String)	250	47,285.38 (19.7%)	100.48 (0.0%)	189.14 (0.1%)	0.40 (0.0%)	254 (0.0%)	1 (0.0%)
AutonomyResultSet.getQuickSummary()	250	28,686.73 (12.0%)	70.34 (0.0%)	114.75 (0.0%)	0.28 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getSummary()	250	25,913.52 (10.8%)	80.38 (0.0%)	103.65 (0.0%)	0.32 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getDatabaseNum()	250	25,672.37 (10.7%)	10.05 (0.0%)	102.69 (0.0%)	0.04 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getWeight()	250	25,240.31 (10.5%)	0.00 (0.0%)	100.96 (0.0%)	0.00 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getID()	250	24,567.10 (10.3%)	60.29 (0.0%)	98.27 (0.0%)	0.24 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getTitle()	250	23,662.79 (9.9%)	160.77 (0.1%)	94.65 (0.0%)	0.64 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getURL()	250	23,431.69 (9.8%)	30.14 (0.0%)	93.73 (0.0%)	0.12 (0.0%)	252 (0.0%)	1 (0.0%)
BufferedReader.<init>(Reader)	2,050	8,972.77 (3.7%)	8,972.77 (3.7%)	4.38 (0.0%)	4.38 (0.0%)	2,050 (0.3%)	1 (0.0%)
Syslog.log(Object, Object, Object, Object, int)	7,302	8,771.81 (3.7%)	60.29 (0.0%)	1.20 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
Syslog.log(InetAddress, Object, Object, Object, Object, int)	7,302	8,711.52 (3.6%)	8,711.52 (3.6%)	1.19 (0.0%)	1.19 (0.0%)	29,284 (4.4%)	4 (0.0%)
AutonomyStatement.executeQuery()	25	5,275.14 (2.2%)	0.00 (0.0%)	211.01 (0.1%)	0.00 (0.0%)	35 (0.0%)	1 (0.0%)

Methods with the most time spent only in that method (not including sub-methods):

Method Time

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
BufferedReader.readLine()	140,625	95,887.04 (40.0%)	95,887.04 (40.0%)	0.68 (0.0%)	0.68 (0.0%)	280,728 (42.5%)	1 (0.0%)
AutonomyResultSet.parseResult(String, int)	2,000	218,190.00 (91.1%)	50,470.57 (21.1%)	109.09 (0.0%)	25.24 (0.0%)	143,758 (21.8%)	71 (0.0%)
StringTokenizer.nextToken()	137,350	49,706.93 (20.7%)	49,706.93 (20.7%)	0.36 (0.0%)	0.36 (0.0%)	137,826 (20.9%)	1 (0.0%)
BufferedReader.<init>(Reader)	2,050	8,972.77 (3.7%)	8,972.77 (3.7%)	4.38 (0.0%)	4.38 (0.0%)	2,050 (0.3%)	1 (0.0%)
Syslog.log(InetAddress, Object, Object, Object, Object, int)	7,302	8,711.52 (3.6%)	8,711.52 (3.6%)	1.19 (0.0%)	1.19 (0.0%)	29,284 (4.4%)	4 (0.0%)
StringBuffer.append(int)	4,225	4,923.47 (2.1%)	4,923.47 (2.1%)	1.17 (0.0%)	1.17 (0.0%)	12,925 (2.0%)	3 (0.0%)
Syslog.configure(File)	1	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	12,655 (1.9%)	12,655 (1.9%)

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
StringBuffer.toString()	8,451	3,366.04 (1.4%)	3,366.04 (1.4%)	0.40 (0.0%)	0.40 (0.0%)	8,451 (1.3%)	1 (0.0%)
StringTokenizer.<init>(String, String)	135,775	2,743.07 (1.1%)	2,743.07 (1.1%)	0.02 (0.0%)	0.02 (0.0%)	0 (0.0%)	0 (0.0%)
StringBuffer.<init>(String)	6,326	1,999.53 (0.8%)	1,999.53 (0.8%)	0.32 (0.0%)	0.32 (0.0%)	6,326 (1.0%)	1 (0.0%)
HttpURLConnection.getInputStream()	50	1,778.48 (0.7%)	1,778.48 (0.7%)	35.57 (0.0%)	35.57 (0.0%)	4,328 (0.7%)	86 (0.0%)
HttpURLConnection.connect()	50	1,356.47 (0.6%)	1,356.47 (0.6%)	27.13 (0.0%)	27.13 (0.0%)	2,681 (0.4%)	53 (0.0%)
StringBuffer.append(String)	13,050	1,065.08 (0.4%)	1,065.08 (0.4%)	0.08 (0.0%)	0.08 (0.0%)	2,300 (0.3%)	0 (0.0%)
StringTokenizer.hasMoreTokens()	135,925	1,014.84 (0.4%)	1,014.84 (0.4%)	0.01 (0.0%)	0.01 (0.0%)	0 (0.0%)	0 (0.0%)
BufferedReader.readLine()	140,625	95,887.04 (40.0%)	95,887.04 (40.0%)	0.68 (0.0%)	0.68 (0.0%)	280,728 (42.5%)	1 (0.0%)

Average Cumulative Time (includes time spent in sub-methods):

Average Cumulative Time

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
Syslog.addLogging()	1	3,536.86 (1.5%)	0.00 (0.0%)	3,536.86 (1.5%)	0.00 (0.0%)	16 (0.0%)	16 (0.0%)
Syslog.configure(File)	1	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	12,655 (1.9%)	12,655 (1.9%)
Syslog.<clinit>()	1	954.55 (0.4%)	20.10 (0.0%)	954.55 (0.4%)	20.10 (0.0%)	30 (0.0%)	30 (0.0%)
Syslog.<clinit>()	1	854.07 (0.4%)	854.07 (0.4%)	854.07 (0.4%)	854.07 (0.4%)	6,570 (1.0%)	6,570 (1.0%)
AutonomyStatement.executeQuery()	25	5,275.14 (2.2%)	0.00 (0.0%)	211.01 (0.1%)	0.00 (0.0%)	35 (0.0%)	1 (0.0%)
AutonomyStatement.execute(String)	25	5,054.09 (2.1%)	10.05 (0.0%)	202.16 (0.1%)	0.40 (0.0%)	137 (0.0%)	5 (0.0%)
AutonomyResultSet.getCustomField(String)	250	47,285.38 (19.7%)	100.48 (0.0%)	189.14 (0.1%)	0.40 (0.0%)	254 (0.0%)	1 (0.0%)
AutonomyResultSet.getQuickSummary()	250	28,686.73 (12.0%)	70.34 (0.0%)	114.75 (0.0%)	0.28 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyStatement.popResults(InputStream)	25	2,733.03 (1.1%)	30.14 (0.0%)	109.32 (0.0%)	1.21 (0.0%)	79 (0.0%)	3 (0.0%)
AutonomyResultSet.parseResult(String, int)	2,000	218,190.00 (91.1%)	50,470.57 (21.1%)	109.09 (0.0%)	25.24 (0.0%)	143,758 (21.8%)	71 (0.0%)
AutonomyResultSet.getSummary()	250	25,913.52 (10.8%)	80.38 (0.0%)	103.65 (0.0%)	0.32 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getDatabaseNum()	250	25,672.37 (10.7%)	10.05 (0.0%)	102.69 (0.0%)	0.04 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyConnection.<init>()	25	2,562.21 (1.1%)	0.00 (0.0%)	102.49 (0.0%)	0.00 (0.0%)	2 (0.0%)	0 (0.0%)

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
AutonomyResultSet.getWeight()	250	25,240.31 (10.5%)	0.00 (0.0%)	100.96 (0.0%)	0.00 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getID()	250	24,567.10 (10.3%)	60.29 (0.0%)	98.27 (0.0%)	0.24 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getTitle()	250	23,662.79 (9.9%)	160.77 (0.1%)	94.65 (0.0%)	0.64 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyResultSet.getURL()	250	23,431.69 (9.8%)	30.14 (0.0%)	93.73 (0.0%)	0.12 (0.0%)	252 (0.0%)	1 (0.0%)
AutonomyConnection.isServerStatusOK()	25	1,688.05 (0.7%)	0.00 (0.0%)	67.52 (0.0%)	0.00 (0.0%)	58 (0.0%)	2 (0.0%)
HttpURLConnection.getInputStream()	50	1,778.48 (0.7%)	1,778.48 (0.7%)	35.57 (0.0%)	35.57 (0.0%)	4,328 (0.7%)	86 (0.0%)
AutonomyConnection.getProperties()	25	864.12 (0.4%)	30.14 (0.0%)	34.56 (0.0%)	1.21 (0.0%)	170 (0.0%)	6 (0.0%)

Average time spent only in a method (not including sub-methods):

Average Method Time

Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Average Method Time	Method Objects	Average Method Objects
Syslog.configure(File)	1	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	3,506.71 (1.5%)	12,655 (1.9%)	12,655 (1.9%)
Syslog.<clinit>()	1	854.07 (0.4%)	854.07 (0.4%)	854.07 (0.4%)	854.07 (0.4%)	6,570 (1.0%)	6,570 (1.0%)
HttpURLConnection.getInputStream()	50	1,778.48 (0.7%)	1,778.48 (0.7%)	35.57 (0.0%)	35.57 (0.0%)	4,328 (0.7%)	86 (0.0%)
HttpURLConnection.connect()	50	1,356.47 (0.6%)	1,356.47 (0.6%)	27.13 (0.0%)	27.13 (0.0%)	2,681 (0.4%)	53 (0.0%)
AutonomyResultSet.parseResult(String, int)	2,000	218,190.00 (91.1%)	50,470.57 (21.1%)	109.09 (0.0%)	25.24 (0.0%)	143,758 (21.8%)	71 (0.0%)
Syslog.<clinit>()	1	954.55 (0.4%)	20.10 (0.0%)	954.55 (0.4%)	20.10 (0.0%)	30 (0.0%)	30 (0.0%)
Properties.load(InputStream)	25	432.06 (0.2%)	432.06 (0.2%)	17.28 (0.0%)	17.28 (0.0%)	1,125 (0.2%)	45 (0.0%)
ClassLoader.loadClassInternal(String)	26	190.91 (0.1%)	190.91 (0.1%)	7.34 (0.0%)	7.34 (0.0%)	702 (0.1%)	27 (0.0%)
BufferedReader.<init>(Reader)	2,050	8,972.77 (3.7%)	8,972.77 (3.7%)	4.38 (0.0%)	4.38 (0.0%)	2,050 (0.3%)	1 (0.0%)
AutonomyStatement.parseText(int, String, boolean)	50	221.05 (0.1%)	130.62 (0.1%)	4.42 (0.0%)	2.61 (0.0%)	100 (0.0%)	2 (0.0%)

7.3 User Guide

7.3.1 Introduction

7.3.1.1 Purpose

This user guide is meant to be a resource for the installation, configuration, and usage of the Reusable Common Services (RCS) Search Framework and its complementary JSP Search Tag Library (Taglib). The RCS Search Framework may be used independently from the Taglib, yet the Taglib implements and requires the underlying Search Framework.

7.3.1.2 Intended audience

People who will benefit the most from this section are ITA and SFA application developers who need to use the RCS Search Framework in their applications. Developers who need to use the JSP Search Taglib also will find this section useful.

7.3.1.3 Background

This Search framework is part of a suite of frameworks called Reusable Common Services (RCS) provided to SFA applications by the ITA initiative. The goal of the ITA initiative is to promote code reuse, standardization, and application of best practices across all SFA system development projects. The Search framework simplifies, standardizes, and improves the use of the Autonomy search engine. The JSP Search Tag Library (Taglib) contains tags that integrate with the RCS search framework and provide basic logic and JSP / HTML functionality. Tag libraries are composed of a set of custom tags. These custom tags help separate presentation from business logic. What this means is that web designers can change the layout without worrying about modifying the underlying logic. Custom tags also help developers avoid embedding scripting code within the JSP page as well as encourage reuse and ease maintainability.

7.3.1.4 Scope

This section covers the installation, configuration and usage of the RCS Search Framework and the RCS Search JSP Taglib. Detailed information about their design may be found in the RCS Search Design document and the RCS Search JSP Taglib Reference.

7.3.1.5 Assumptions

We assume the following environment:

- J2EE application server: WebSphere application server
- JSP 1.1 Processor configured (See Appendix B of the RCS Search JSP Taglib Reference)
- Java Servlet 2.2
- Reader is familiar with, if not proficient in, JSP development

7.3.2 Description

7.3.2.1 Overview

The search framework simplifies, standardizes, and improves the use of the Autonomy search engine. This framework complies with J2EE standards instead of using CGI as in the current search engine interface. The framework consists of classes that provide a common way to access the Autonomy HTTP API and utilize its features.

The search framework implements the following Autonomy features:

- Query search engine (including querying a custom field)
- Natural Language or "Fuzzy" query search engine (including querying a custom field)
- Display search results (including sorting by a custom field)
- Suggest additional search results

The RCS Search JSP Tag Library, or Search Taglib, defines a set of custom tags for use in Java Server Pages (JSPs), which provide a number of benefits. Amongst the benefits of using the custom tags are the following:

- Developer friendly interface to RCS Search core components
- Java code is removed from JSPs
- Common tasks can be delegated to the taglib

The primary responsibilities of the Tag Library are to:

- Provide the custom tags for use in JSPs
- Map the tags to their corresponding tag classes
- Give usage guidelines, including required and optional attributes.

These custom tags are not just bean tags. Custom tags can modify the content within the tag body and have access to the application context. Some of the ways they can be used include dynamically generating page content and implementing flow of control. They can interact with each other including being nested.

7.3.2.2 RCS Search Framework Features

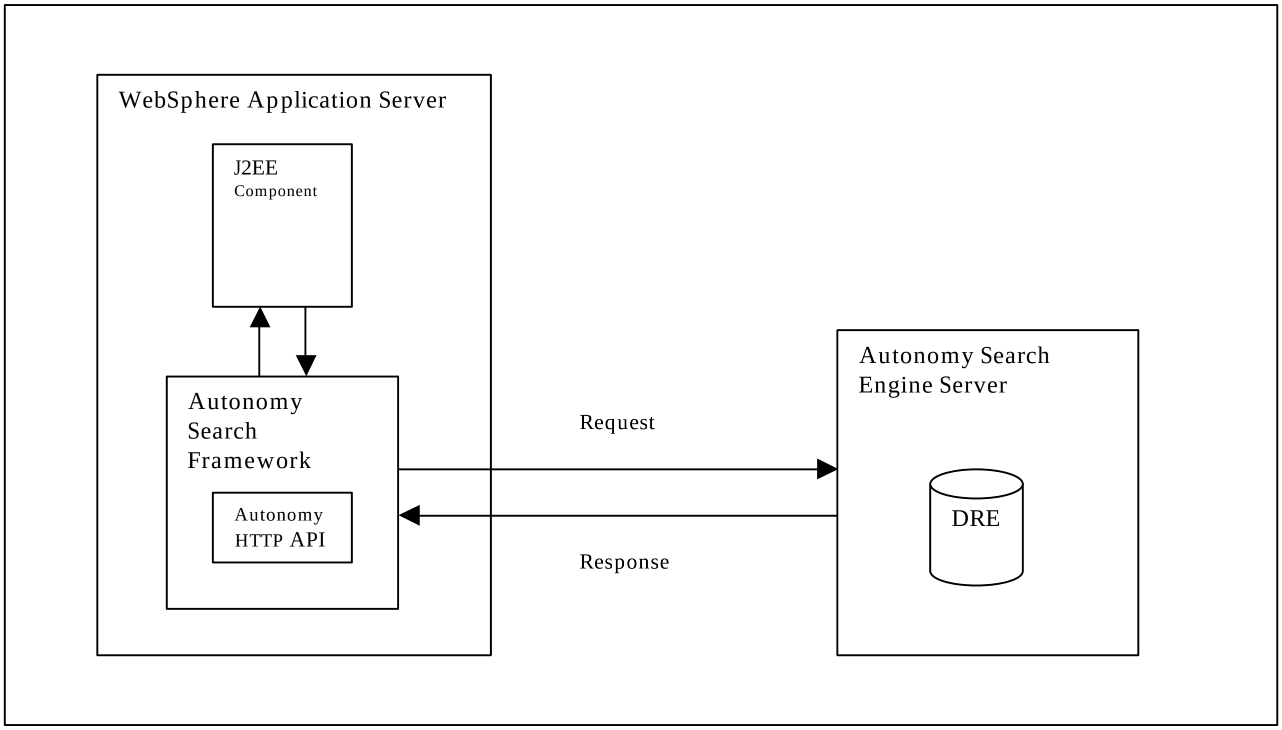
The Search framework consists of three classes listed below:

- AutonomyConnection
- AutonomyStatement
- AutonomyRestultSet

These classes wrap the underlying HTTP API commands provided by the Autonomy Search Engine server. The AutonomyConnection class represents the Autonomy Search Engine Server connection information. The AutonomyStatement class represents statements sent to the Autonomy Search Engine server. The AutonomyResultSet class represents results returned from the Autonomy Search Engine server.

For exception handling and logging in this search framework, the ITA R2.0 Exception and Logging frameworks will be used. These powerful frameworks will ensure that sufficient exception handling and logging are included with this search framework.

Below is a diagram that illustrates the interaction between any J2EE component (JSP, Servlet, EJB, JavaBean), the Search framework, and the Autonomy Search Engine server.


WebSphere Application Server
J2EE Component
Autonomy Search Framework
Autonomy HTTP API
Request
Response
Autonomy Search Engine Server
DRE

7.3.2.3 RCS Search JSP Taglib Features

The Search JSP Taglib provides tags for the following Search Components, outlined in the RCS Search Framework Design Document, and described above:

- Connection : Defines an AutonomyConnection object
- Statement : Defines an AutonomyStatement object with Connection parameters
- ResultSet : Executes Statement to create an AutonomyResultSet object

Instructions on how to use these tags are in the RCS Search JSP Taglib Reference document.

7.3.2.4 Main Concepts

The Search framework allows the programmer to execute searches against the Autonomy server through simple API. Instead of learning Autonomy's HTTP-based API, the developer can easily use the methods available through the RCS Search framework. A Search JSP Taglib is provided to aid the developer even more. With this powerful set of JSP 1.1-based tags the programmer can rapidly develop JSP pages to query the Autonomy server.

7.3.3 Installation

7.3.3.1 Software requirements

The RCS Search framework is J2EE compliant. The framework requires JDK 1.2 (recommended) and also works with JDK 1.3.

The following is a useful installation table of requirements. Use this to make sure you have all of the required files and applications in your environment.

Operating System	The ITA Team has tested the search framework in the following operating systems: Solaris 2.6, Windows NT/2000
Application Server Environment	The ITA Team has tested the search framework within WebSphere Application Server 3.5. There is nothing in the search package that ties the framework to a particular application server (in addition, minimal testing has been completed on the Jakarta-Apache base). Also, a JSP 1.1 Processor is required for using the RCS Search Taglib.
ITA Search Package (rcs_search-v1.1.jar)	jakarta-oro-2.0.1.jar jdom-B6.jar protomatter-1_1_5.jar rcs_exception-v1.5.jar rcs_logging-v1.5.jar utility.jar xerces.jar xml.jar
ITA Search Taglib Package (rcs_search-taglibs-v1.1.jar)	search.tld – RCS Search Tag Library Descriptor file rcs_search-v1.1.jar (required for use with rcs_search-taglibs-v1.1.jar) javax.servlet.jsp.* (especially javax.servlet.jsp.tagext.*)
Properties Files	autonomy.properties errorMessages.properties

7.3.3.2 Installation procedures

Because the RCS Search framework is independent from the Search JSP tag library, this documentation will proceed with the base installation/configuration for the RCS Search framework followed by additional configuration steps necessary to install/configure the Search JSP Taglib.

7.3.3.2.1 RCS Search Framework

Place the above .jar files in a directory not accessible from the web on your web/application server. This directory/path will eventually (if not already) be listed in the application server's classpath.

The search framework binaries are located in `rcs_search-v1.1.jar.`

7.3.3.2.2 RCS Search JSP Taglib

Follow the guidelines and procedures outlined in the RCS Search JSP Taglib Reference document for setting up the tag library. The properties files need to be located in the root directory.

The search tag library binaries are located in `rcs_search-taglibs-v1.1.jar.`

7.3.4 Configuration

7.3.4.1 Add the Jar files on the classpath

The search package needs to be added in the WebSphere classpath. The following steps show how to add the classpath on WebSphere:

- Bring up the WebSphere admin console and select your application server on the console
- Stop your application server
- Click on the 'General' Tab and add the following line in the Command Line Arguments **-classpath /www/dev/rcs/jars/ (example for development environment)**
- Restart your application server.

7.3.4.2 Add the StartupRcs.jar on the classpath

Because the Logging framework is used by the Search framework, it is required to set up the Logging framework correctly by following the steps in this section.

Due to the fact that multiple applications can use the same ITA Reusable Common Services (RCS), it is beneficial to configure and launch a startup class that configures and starts any ITA RCS Services within an Application Server. This is accomplished by using WebSphere's ServiceInitializer interface. By specifying the name of the Startup Class as part of the ServiceInitializer command line argument for the Application Server, WebSphere will run the class as the last action it does in an Application Server startup or shutdown. An example is:

- `Dcom.ibm.ejs.sm.server.ServiceInitializer=<class>[,<class>]...`

… where each <*class*> is one of the startup classes.

An example at SFA of this is the ITA Logging service. The logging service actually configures itself to the parameters specified within an XML configuration file (rcs.xml). The ITA RCS Startup class reads the XML file and configures the service upon startup of the Application Server. The StartupRcs class code can be found on the 'Sample code' section of this document.

To enable the RCS startup class the following steps need to be taken.

3. The StartupRcs.jar file needs to be placed within the classpath of the Application Server:
   - Bring the WebSphere Admin console up and select your application server.
   - Stop your application server.
   - Click on the 'General' Tab and add the path where StartupRcs.jar is located in the Command Line Arguments.
   - Restart your application server

4. Add the following line in the Command Line Argument

- `Dcom.ibm.ejs.sm.server.ServiceInitializer=gov.ed.sfa.ita.common.StartupRcs`

7.3.4.3 XML configuration file

The Search framework implements the RCS Logging framework, which is configured by the XML configuration file. Please refer to the ITA RCS Logging User Guide for more instructions on setting up the XML configuration file.

7.3.5 Usage Scenarios with Sample Code

Three examples are provided to illustrate the use of the Search framework and the Search JSP Taglib. First, a simple class example is used to illustrate basic usage. The second example (section 5.2) shows a Servlet processor. The third example (section 5.3) is a simple JSP form using the custom search tag library. The ITA team recommends using the RCS Search JSP Taglib because it's easy to use and allows for rapid development.

7.3.5.1 Search Framework with Simple Class

```
package gov.ed.sfa.ita.samples;

import gov.ed.sfa.ita.exception.*;
import gov.ed.sfa.ita.search.*;

public class AutonomyTest {
      /**
       * AutonomyTest constructor comment.
       */
      public AutonomyTest() {
            super();
      }

      public static void main(String[] args) {
            try {
                  AutonomyConnection aCon = new AutonomyConnection();

                  AutonomyStatement aStmt = aCon.createStatement();
                  aStmt.setQueryText("IFAP");
                  //aStmt.setMaxNumResults(10);
                  aStmt.setDatabaseNums("34");
                  aStmt.setQueryTextCustomField("07/19/2001", "Posted_Date");

                  //AutonomyResultSet aRS = aStmt.executeQuery();
                  //AutonomyResultSet aRS = aStmt.executeFuzzyQuery();

                  aStmt.setID("120626");
                  AutonomyResultSet aRS = aStmt.executeSuggest();

                  System.out.println("Total Results=" + aRS.getNumResults());
                  do {
                        System.out.println("Result " + aRS.getRow() + " of " +
aRS.getNumResults());
```

```
                        System.out.println("---------------------------------------
----------------");
                        System.out.println("Title::" + aRS.getTitle());
                        System.out.println("Summary::" + aRS.getSummary());
                        System.out.println("QuickSummary::" +
aRS.getQuickSummary());
                        System.out.println("URL::" + aRS.getURL());
                        System.out.println("Weight::" + aRS.getWeight());
                        System.out.println("ID::" + aRS.getID());
                        System.out.println("DB Num::" + aRS.getDatabaseNum());
                        System.out.println("Posted_Date::" +
aRS.getCustomField("Posted_Date"));
                        System.out.println();

                  } while (aRS.next());

            } catch (SFAException e) {
                  System.out.println(e.getMessage());
            }
      }
}
```

7.3.5.2 Search Framework with Servlet

SearchProcessor.java

```
package gov.ed.sfa.ita.samples;

import gov.ed.sfa.ita.search.*;
import gov.ed.sfa.ita.exception.*;
import java.io.*;

public class SearchProcessor extends javax.servlet.http.HttpServlet {

      public void destroy() {
      }

      /**
       * Process incoming HTTP GET requests
       *
       * @param request Object that encapsulates the request to the servlet
       * @param response Object that encapsulates the response from the
servlet
       */
      public void doGet(
            javax.servlet.http.HttpServletRequest request,
            javax.servlet.http.HttpServletResponse response)
            throws javax.servlet.ServletException, java.io.IOException {

            performTask(request, response);

      }
      /**
       * Process incoming HTTP POST requests
       *
       * @param request Object that encapsulates the request to the servlet
       * @param response Object that encapsulates the response from the 
servlet
       */
      public void doPost(
            javax.servlet.http.HttpServletRequest request,
            javax.servlet.http.HttpServletResponse response)
            throws javax.servlet.ServletException, java.io.IOException {

            performTask(request, response);

      }
      /**
       * Returns the servlet info string.
       */
      public String getServletInfo() {

            return super.getServletInfo();

      }
      /**
       * Initializes the servlet.
       */
      public void init() {
      }
      /**
       * Process incoming requests for information
       * 
       * @param request Object that encapsulates the request to the servlet
       * @param response Object that encapsulates the response from the 
servlet
       */
      public void performTask(
            javax.servlet.http.HttpServletRequest request,
            javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException {

            PrintWriter out = null;
            AutonomyConnection aCon = null;
            AutonomyStatement aStmt = null;
            AutonomyResultSet aRS = null;

            try {

                  aCon = new AutonomyConnection();

                  aStmt = aCon.createStatement();
                  String max = request.getParameter("MaxNumResults");
                  if (max != null && !max.equals("")) {
                        aStmt.setMaxNumResults(Integer.parseInt(max));
                  }
                  String min = request.getParameter("MinThreshhold");
                  if (min != null && !min.equals("")) {
                        aStmt.setMinThreshold(Integer.parseInt(min));
                  }


      aStmt.setDatabaseNames(request.getParameter("DatabaseNames"));
```

```
                  aStmt.setDatabaseNums(request.getParameter("DatabaseNums"));
                  aStmt.setID(request.getParameter("DocID"));

                  String searchtype = request.getParameter("SearchType");
                  if (searchtype.equals("regular")) {
                        //do nothing this is the regular search type
                  } else
                        if (searchtype.equals("booleanand")) {
                              aStmt.setQueryAllWords();
                        } else
                              if (searchtype.equals("booleanand")) {
                                    aStmt.setQueryAnyWords();
                              }

                  aStmt.setQueryText(request.getParameter("QueryText"));
                  aStmt.setQueryTextCustomField(
                        request.getParameter("CustomQueryText1"),
                        request.getParameter("CustomQueryFieldName1"));
                  aStmt.setQueryTextCustomField(
                        request.getParameter("CustomQueryText2"),
                        request.getParameter("CustomQueryFieldName2"));

      aStmt.addCustomQueryStmt(request.getParameter("CustomQueryStmt"));

                  String actiontype = request.getParameter("ActionType");
                  if (actiontype.equals("query")) {
                        aRS = aStmt.executeQuery();
                  } else
                        if (actiontype.equals("fuzzyquery")) {
                              aRS = aStmt.executeFuzzyQuery();
                        } else
                              if (actiontype.equals("suggest")) {
                                    aRS = aStmt.executeSuggest();
                              }

                  response.setContentType("text/html");
                  out = new PrintWriter(response.getOutputStream());
                  out.println("<HTML><BODY>");

                  out.println("<B>Total Results=</B>" + aRS.getNumResults());
                  out.println("<BR>");
                  do {
                        out.println("<B>Result</B> " + aRS.getRow() + " of " +
aRS.getNumResults());
                        out.println("<BR>");
                        out.println("<B>--------------------------------------
------------</B>");
                        out.println("<BR>");
                        out.println("<I>Title::</I>" + aRS.getTitle());
                        out.println("<BR>");
                        out.println("<I>Summary::</I>" + aRS.getSummary());
                        out.println("<BR>");
                        out.println("<I>QuickSummary::</I>" +
aRS.getQuickSummary());
                        out.println("<BR>");
                        out.println("<I>URL::</I>" + aRS.getURL());
                        out.println("<BR>");
```

```
            out.println("<I>Weight::</I>" + aRS.getWeight());
            out.println("<BR>");
            out.println("<I>ID::</I>" + aRS.getID());
            out.println("<BR>");
            out.println("<I>DB Num::</I>" + aRS.getDatabaseNum());
            out.println("<BR>");
            out.println("<I>Posted_Date::</I>" +
aRS.getCustomField("Posted_Date"));
            out.println("<BR>");
            out.println("<BR>");
         } while (aRS.next());
         out.println("</BODY></HTML>");
         out.close();
      } catch (SFAException e) {
         if (out == null) {
            out = new PrintWriter(response.getOutputStream());
            out.println("<HTML><BODY>");
         }
         out.println("<B>ERROR</B>");
         out.println("<BR><I>");
         out.println(e.getMessage());
         out.println("</I>");
         out.println("</BODY></HTML>");
         out.close();
         System.out.println(e.getMessage());
      } catch (Exception e) {
         if (out == null) {
            out = new PrintWriter(response.getOutputStream());
            out.println("<HTML><BODY>");
         }
         out.println(e.getMessage());
         out.close();
         System.out.println(e.getMessage());
      }
   }
}
```

7.3.5.3 Search JSP Taglib with JSP

```
<%@ page language="java" %>
<%@ taglib uri="/search-taglib" prefix="search"%>
<html>
<head>
<title>Search Results Example</title>
</head>
<!--This is an example of how to use the RCS Search custom JSP Tag Libraries -
->

<body>

<!-- This sets up the connection to the Autonomy Server-->
<search:connection id="conn1"/>

<!--
// This builds the query statement
// Here, if this were a form processor, we would get the
// posted form variables.
```

```
-->
<search:statement id="stmt1" conn="conn1">
   <search:query queryText="Illinois" maxNumResults="40"
dbNames="eannouncements">
       <search:customquery queryText="07/19/2001" fieldName="Posted_Date"/>
   </search:query>
</search:statement>

<!-- Start building the results and the table display -->
<search:resultset stmt="stmt1" shouldLoop="false" >
   <B><search:getTotalNumResults/> result(s) for query
'<search:getQueryText/>'<B>
</search:resultset>

<table border=1>
<search:resultset stmt="stmt1" shouldLoop="true" startRow="1" endRow="max" >
<tr>
   <td>Result <search:getResultNum/> of <search:getTotalNumResults/></td></tr>
<tr>
   <td><B>Title</B></td> <td><A
HREF="<search:getURL/>"><search:getTitle/></A></td> </tr>
<tr>
   <td><B>Summary</B></td> <td><search:getSummary/></td> </tr>
<tr>
   <td><B>Quick Summary</B></td> <td><search:getQuickSummary/></td> </tr>
<tr>
   <td><B>Weight</B></td> <td><search:getWeight/></td> </tr>
<tr>
   <td><B>ID</B></td> <td><search:getID/></td> </tr>
<tr>
   <td><B>DB Num</B></td> <td><search:getDBNum/></td> </tr>
<tr>
   <td><B>Custom Field</B></td> <td><search:getCustomField
fieldName="Posted_Date"/></td> </tr>
</search:resultset>
</table>

</body>
</html>
```

7.3.6 Resources

The following resources have more information about Autonomy, the Search framework and JSP tag libraries :

- RCS Search Design document
- RCS Search JSP Taglib Reference document
- http://jakarta.apache.org/taglibs/tutorial.htm